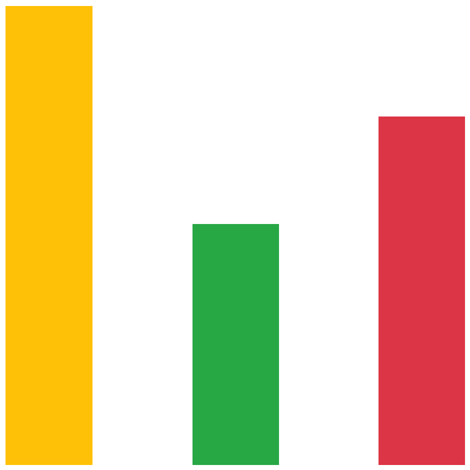
SentiCloud

A platform for social media sentiment analysis

Project Report

Group SW704E18



AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø
http://cs.aau.dk

Title:
SentiCloud: A Social Media Analytics Platform

Theme:
Internet Technology

Project Period:
Fall Semester 2018

Project Group:
sw704e18

Participant(s):
Anders Højlund Brams
Anders Langballe Jakobsen
Matias Røikjær Jensen
Mikkel Jarlund
Theis Erik Jendal
Thomas Buhl Andersen

Supervisor(s):
Peter Dolog

Page Count: 130

Date of Completion:
December 18, 2018

Abstract:

Context is important, and while sentiment analysis is a useful tool it can be difficult to use effectively if all you have is a numerical sentiment score without any context. In an attempt to make sentiment analysis easier to decipher, this report introduces SentiCloud, a system capable of providing sentiment statistics along with a more descriptive context in the form of keywords.
SentiCloud is capable of continuously gathering information from social media platforms concerning user specified brands at an arbitrary granularity of sub-topics. By using a combination of deep neural networks for sentiment analysis and graph based keyword extraction, SentiCloud can provide a user with sentiment statistics and contextual keywords through a UI designed in accordance with the Material Design language.
With a service oriented architecture, SentiCloud is designed to be able to scale vertically and horizontally in terms of both computational power and storage. Through a large-scale stress test of load tolerance we show that with appropriate scaling SentiCloud can handle large quantities of data and user interactions allowing it to deliver a high quality service to a large user base.

Preface

This report covers the development of a system capable of conducting sentiment analysis over tweets regarding brands and topics and groups significant keywords- and phrases by their sentiment. The project is conducted in accordance with the study curriculum for 7th semester software engineering students at Aalborg University that requires the design and construction of a web application that is scalable, functional, and solves a specific problem.

The methods and techniques used during this development process will be described in chapter 2.

The constructed system solves the problem of determining the source of positive and negative sentiment behind tweets regarding a brand or topic. This problem is analysed and identified in chapter 3 and formulated precisely as the problem definition in section 3.4.

Chapter 4 will further define the requirements for our application, bot functional, section 4.1 and non-functional, section 4.2.

In appendix A we will explore various machine learning architectures and keyword extraction techniques that were utilised in the development of our system.

The process of designing the system is covered in chapter 5 and its implementation is covered in chapter 6. The system is evaluated and compared to a set of baseline solutions in chapter 7 and the evaluation results are summarised and discussed in chapter 8.

Throughout the report, it is assumed that the reader has a basic understanding of probability theory as well as basic principles of artificial neural networks, including backpropagation.

Contents

Todo list

Chapter 1: Introduction

Historically, humans have relied on the opinions of friends and family in order to make non-trivial decisions (e.g. what products to purchase and who to vote for in a political election) [31]. With the development of the social media, one is not limited to friends and family anymore. Consumers often use the opinions of other people before buying products. A business therefore might not need to conduct surveys and polls to get an understanding of the market, but can rely on the posts and opinions expressed on the internet. By analysing social media using Sentiment Analysis (SA), it is possible to predict stock prices, election results and box office revenue [31]. Furthermore, a business can use a SA tools as a means of reputation management. If the opinion about some business is shifting in a certain direction, a business can use SA to observe the shift and act accordingly. Companies are increasingly noticing this influence and significance their customers have over the opinions read on the web and therefore what a customer might buy [54]. The ability to capitalise on the opinions of the consumers Social Media Posts (SMPs) is therefore important for a business.

Applying SA to SMPs on its own is a useful tool for reputation management. However, both real-time and historical SA on their own offer no insight into the polarity of individual subtopics within some brand. For example, one could imagine a brand with an overall equal distribution of positive and negative mentions. While it might simply be the case that the brand is generally perceived with mixed sentiments, it could also be the case that certain topics relating to the brand have unbalanced distributions of positive or negative mentions. Therefore, being able to see the common phrases and words used in SMPs with similar sentiment would provide a useful insight.
Given the evident importance of SA and Keyword Extraction (KE), we present the following initial problem statement:

> Is it possible to determine user intent regarding brands and topics in a more descriptive way than binary sentiment classification?

The use of social media has become a popular way for businesses to gain business value. Businesses can use social media to promote its brand which, if successful, adds business value in the form of customer traffic, loyalty and retention [15]. Businesses have always been able to run promotional campaigns in the form of newspaper ads and the likes. The interesting part about running promotional campaigns on social media is the fact that customer sentiments can be gathered and analysed.

Negative comments from social media, like Facebook and Twitter, can harm the brand and be used by competitors to gain new customers. Reputation management can be used to monitor the image of the brand or business. Because of the impact social media has on brand reputation, it is desirable to monitor the sentiment and opinions of comments on social networks [19].

Chapter 2: Development process

In this chapter we describe the various development methods, practices, and hardware used for this project.

2.1 Scrum

Based on our previous experiences of applying Scrum we will selectively be using some of the practices of Scrum. This section lists these practices, our experiences with them and how we will be using them.

2.1.1 Daily scrum/Stand-up meeting

We will strive to implement a daily stand-up meeting in order to keep track of work progress and any issues that may arise so as to quickly rectify them. Our experiences with holding a daily stand-up meeting suggests that we cannot hold it at the start of the day due to people being absent. Delaying the meeting until everyone is present has not worked for us as the ones that met on time were by then already engaged in their tasks and found the interruption disruptive. As such planning these meetings at the start of the day might not be optimal. An alternative might be to hold the meeting later, while avoiding interrupting people 'in the zone'. For this, two options present themselves: either hold the meeting in connection with the lunch break, or at the end of day. After weighing the options we have chosen to still try holding the meeting at the start of the day. This is due to people feeling that the early meeting serves as a good way to get people engaged in the work. To address the issues we had previously we will try to be more rigid with the scheduling of the meeting, having it at 9:30 regardless of any delays.

2.1.2 Scrum board

We have previously tried to use a Scrum board, but have had trouble keeping with it for the entire project. This was, at least partially, due to the overhead associated with creating or updating tasks in the tool we used[1]. This year we will try to use a Kanban board in GitHub, which should reduce the overhead and thus hopefully make it easier to keep updated throughout the project.

[1]See: `https://www.pivotaltracker.com`.

2.1.3 Scrum sprints

We have previously worked in one month sprints, and while we generally feel that the sprint structure is useful, we feel that a month is too long. As we work with technologies previously unknown to us, there is a lot of uncertainties and rapid changes that make it difficult to plan an entire month at a time. We will thus try doing doing shorter sprints of two weeks at a time. We hope that the shorter sprints will better fit the dynamic nature of our project.

2.2 Test-driven development

We have previously employed Test-Driven Development (TDD) to good effect, and intend to do so again. The essence of TDD is writing both integration- and unit tests before and during implementation of the functionality of the system. This requires us to determine what each function should and should not do when given specific input, and requires us to reason about the prevention and handling of error-prone edge cases. By doing this, we will gradually build up a regression test suite, which removes the need for writing tests after implementation, which we from experience know to be a long and arduous process. With a large regression test suite, we will be able to maintain and modify our code with good confidence that the changes work as expected. If they do not, we will be notified by having some of our regression tests fail.

2.3 Risk Management

We have previously tried to do formal Risk Mitigation, Monitoring and Management planning, but feel that the overhead associated with this process outweighed its usefulness. We will thus keep the risk management more informal, through our daily meetings and sprint evaluation and planning.

2.4 Development server

During development we have used a shared server for deploying the various services. The relevant specifications of the server are as follows:

Hardware

- Motherboard: Hewlett-Packard 1495
- CPU: Intel Core I7-2600 Quad core @ 3.4GHz
- RAM: 4x4GB DIMM DDR3 Synchronous 1333MHz
- Disk: 500GB Seagate ST3500413AS 6Gb/s (HDD)

Software

- OS: Linux, Debian 9 Stretch[2]

[2]See: `https://www.debian.org/releases/stretch/`.

- Docker: 18.09.0[3]
- Docker-compose: 1.8.0[4]

Docker images used

- Tensorflow Serving: bitnami/tensorflow-serving:1.12.0[5]
- Postgresql: postgres:11.1-alpine[6]
- Python: python:3.6.7-slim[7]
- Nginx: nginx:1.15.7-alpine[8]

For a quick read about our server setup, see appendix B.

[3]See: `https://docs.docker.com/`.
[4]See: `https://docs.docker.com/compose/`.
[5]See: `https://hub.docker.com/r/bitnami/tensorflow-serving/`.
[6]See: `https://hub.docker.com/_/postgres/`.
[7]See: `https://hub.docker.com/_/python/`.
[8]See: `https://hub.docker.com/_/nginx/`.

Chapter 3: Problem analysis

In this chapter we will conduct an analysis of the problem domain. Initially, we will briefly look into methods for conducting user opinion analysis in section 3.1 and section 3.2. Afterwards, we discuss the necessity for the resulting system to be both scalable and tolerant to heavy computational loads in section 3.3. Although we will not delve into the models necessary in the development of such a platform, they will still be touched upon and then further described in appendix A and chapter 5. The result of the analysis is a problem definition in section 3.4, which will serve as a basis for the development of the system.

3.1 Sentiment analysis

This section will briefly cover SA as an approach for determining user intent and opinions from social media. Social media platforms like Twitter, Facebook, and Reddit are saturated with platform users stating their thoughts and opinions regarding a brand or topic through text. Unfortunately, unlike typical product reviews with a rating system, the positive or negative attitude towards a brand is rarely explicitly stated in SMPs, but rather indicated implicitly. For a system to be able to determine whether a SMP carries a positive or negative sentiment towards an entity, we need a method for computationally determining sentiment or intent from the text alone.

Initial research shows that several approaches exist for this, all varying in their underlying principles, computational complexity and accuracy. Although the approaches differ in how they achieve the same goal, they all use Natural Language Processing (NLP) techniques and text analysis in different ways to determine the general sentiment of a piece of text. SA can be divided into two general subcategories: lexical/knowledge-based analysis and Machine Learning (ML)-based methods [20, p. 27]. We will explore the different approaches more closely in the sections following the problem definition to be stated in section 3.4.

3.1.1 Existing solutions

We will now take a look at existing solutions, in order to get an overview of how other solutions group topics based on sentiment and discussion of brands. This can be a difficult task as commercial solutions often do not describe how their solutions work but only what purpose the solution has. Assessing whether or not they use a certain feature or approach can be difficult without having a license to use the software. The comparison will mainly be focused on features relating to the initial problem statement in chapter 1.

Brand24

Brand24 offers social insights through the use of SA. Where Brand24 might differ from other solutions, is the fact that they assign an influencer score to people to sort mentions into degrees of influence. That way they have the ability to sort out less influential mentions which may be beneficial to the customer as they pay for an amount of mentions per month. They group results related to specific topics, keywords and word-clouds which may improve their ability to decide whether a mention is positive or negative. Additionally they offer analysis of competitors in order to show costumer reputation compared to a competitor.

Brandwatch

Brandwatch offers reputation management services for various social media. In relation to the features we are seeking to implement, Brandwatch offers analytics tools for SA and trending topic analysis. However, as far as we can tell the topic analysis does not include an option for determining the sentiment for individual topics, i.e. the SA applies to the brand as a whole.

BrandsEye

BrandsEye offers opinion mining on social media with a combined algorithmic and manual approach. The combination of AI and human involvement allows accurate analysis of messages including sarcasm, slang or mixed sentiments. BrandsEye services are, according to them, capable of telling not just how people feel, but what's making them feel that way at a 99% level of confidence with an error margin of 1.75% [3].

Comparison and shortcomings

The three existing solutions mentioned in this chapter all provide paying clients with a SA platform allowing its users to keep track of people's overall sentiment towards a specific brand. Being able to determine the users' sentiment towards a brand is extremely useful for marketers and community managers. However, we see a glaring issue with the existing solutions; user opinion analysts can not work efficiently from just knowing the overall sentiment of their customers; they need to know *why* the sentiments are positive or negative. While Brandwatch has support for brand-wise trending topic analysis, it would be pertinent to support trending topic analysis at an arbitrary level of brand-granularity - for example if the community manager of Google wants to know what people think about their newest phone specifically.

3.2 Extraction of keywords- and phrases

As stated in the analysis of existing solutions for user opinion analysis, subsection 3.1.1, just knowing the sentiment of the users is not enough for user opinion analysts to be efficient in their decision making. It would be pertinent for a user opinion analysis platform to supply the analyst with more contextual information regarding the users' sentiment - for example, in form of a trending topic analysis as provided by Brandwatch.

orpus ændres til
ents?

There are many ways of determining trending topics from a corpusof SMPs. One way is to extract keywords and phrases from the corpusthrough a process conveniently named KE. Several KE methods exist, with approaches ranging from relatively simple approaches identifying keywords from their statistical properties in the text, to ML and graph-based approaches. As we will see in section 5.8, all of these processes are computationally complex in their nature, and as we will see in the following section, the system to be built in this project needs to be scalable and tolerant to heavy data loads. As such, the choice of approach in this regard must be chosen carefully. The aforementioned approaches for KE will be covered in depth in section 5.8.

Same as above

documents?

3.3 Scalability and load tolerance

The quality of the results of user opinion analysis and topic extraction is fundamentally dependent of the overall number of individual opinions. Furthermore, as mentioned in subsection 3.1.1, it is pertinent for a user opinion analyst to analyse the sentiment towards a brand or company at an arbitrary level of sub-brand granularity. For example, the overall sentiment towards Google might not rely on the sentiment towards Google specifically, but likely also the sentiment towards Google-specific products such as phones, websites, and applications. An aggregation over the sentiment towards these sub-topics may yield a more realistic picture of the sentiment towards Google, and could support the ability to discern which sub-topics lead to the largest increase or decrease in overall sentiment.

As the amount of sub-topics and user opinions grows, such a system will naturally incur complications w.r.t. processing all the data in a timely manner. An important aspect of a usable application is that it is responsive. If a user-triggered event takes more than 1 second to process, the sense of flow in performing the task is lost and the system will feel sluggish. If the event takes more than 10 seconds, user attention is lost and will need to be regained upon task completion, that is, if the user does not leave prematurely [38, ch. 5].

Part of this semester's study curriculum states that the presented solution must be a web application that employs a scalable architecture and provides a good quality of service. The quality of the service provided by our system relies, in part, on the responsiveness of the system. Several of the approaches that can be used for user opinion analysis (see section 3.1 and section 3.2) are computationally complex both in terms of the time and space required for performing the analysis (this is expanded upon in appendix A). Such a system must be able to scale both in terms of storage space and computational power in such a way that the system stays responsive despite large amounts of opinion data. As such, while designing our solution, appropriate methods should be chosen from a trade-off between the quality of the results and the computational complexity of the process, and the system itself should be structured in a way that allows for scaling of storage space and computational power.

3.4 Problem definition

The initial problem statement, presented in chapter 1, motivates the ability to determine a more descriptive notion of user sentiment towards a company or brand than a simple binary classification of sentiment. The analysis of existing solutions presented in subsection 3.1.1 found good candidates for how additional information, such as trending topic analysis, could be included in order to make the users' opinions more clear. However, simply determining trending topics for a single company or brand potentially suffers from the problem of being too general. Instead, one could imagine that a system where user opinion analysts could determine the sentiment, keywords and phrases regarding a brand at any arbitrary level of granularity of "sub-brands" or "sub-topics" could help such analysts conduct reputation management in a more reliable way. With a finer level of granularity, it stands to reason that the system will have more data to process and aggregate, as many sub-topics may influence the overall sentiment towards a single entity. The potential solutions explored in section 3.1 and section 3.2 are complex both in terms of space and time. Any system that processes a large amount of data, while needing to be responsive to its users, must be built in a scalable way. Not only must the architecture support outwards scaling in terms of hardware, but the system itself should scale appropriately as the amount of data grows with as little hardware extensions as possible.

Following this problem analysis, we present the following problem definition:

"How can we design and build a scalable and load tolerant system supporting a web application capable of providing system users with meaningful sentimental and topical data regarding customers' opinions towards a company or brand at an arbitrary granularity of sub-brands?"

A complete solution to the defined problem relies on the fulfilment of several system requirements. These are described in chapter 4.

Chapter 4: Requirements

In this chapter we identify and describe the functional and non-functional requirements based on the problem definition in section 3.4. The functional requirements are described in section 4.1 and the non-functional requirements are described in section 4.2. As we are not developing this platform for any particular client, the requirements are mainly ones that we believe will bring value to potential users of the system. Even though we do not have a client we still believe that analysing and prioritising requirements helps us in establishing a guideline for the development of the system. In addition, we believe that the review of existing solutions we have made in subsection 3.1.1 helps us in eliciting requirements, as the publicly displayed functions of these platforms are presumably ones that they believe bring business value to stakeholders.

4.1 Functional requirements

The functional requirements are split in two categories: requirements for the analytical module of the system, and requirements for the overall application and interface. The requirements are identified from the problem definition in section 3.4 as well as this semester's study curriculum.

4.1.1 Analytical module requirements

1. The system must be able to determine the sentiment and keywords from a set of SMPs regarding a specific brand or topic.
2. The system must be able to group and summarise keywords based on assigned sentiment of SMPs.
3. The system must be able to scale and handle large amounts of data and queries without becoming unusable.
4. The system should provide an easy-to-use Application Programming Interface (API) allowing separate client systems to use it, including but not limited to our own web application.
5. The system should dynamically gather and analyse opinion-rich SMPs regarding all relevant brands and topics in the system.

reformulate? Sho
matched in discus

4.1.2 Web application requirements

1. The system must be able to comprehensibly visualise analysis results to a user.
2. The system must be designed in accordance with commonly approved user-interface design

patterns.
3. The system must store historical analytical data such that the user can view changes in sentiment and topics over time.
4. The system must allow users to track multiple brands or topics with associated search keywords.
    (a) The system should have a login system, with each user being assigned a profile linked to their tracked brands.
5. The system should allow users to compare the development in sentiment for different brands.
6. The system must allow users to specify different search keywords for the brands they are tracking.

4.2 Non-functional requirements

In this section we will list non-functional requirements which we wish to fulfil. For each requirement, we will argue why it is necessary to fulfil as well as suggest how we want to fulfil the requirement. We focus on the following non-functional requirements:

- Scalability
- Robustness
- Performance
- Usability
- Testability
- Privacy

4.2.1 Scalability

The primary non-functional requirement of our system is scalability. Scalability is a desirable attribute for many systems and networks, and is characterised by the ability to accommodate an increase in elements to process gracefully. On the other hand, a system is unscalable if there are excessive costs associated with an increased workload or if the system simply cannot cope at the increased levels at all [6]. In the case of our system, examples of increases in elements to process can be an increase in number of SMPs or an increase in traffic from users. Although it may seem that scalability is primarily concerned with load scalability, there is also the issue of space scalability, which is of particular relevance to our system and a factor to account for when designing a storage strategy. In summation, there are aspects to quantifying scalability and all these aspects have to be considered. The following requirements are directly related to the study curriculum requirement of a scalable system (see section 3.3).

To ensure scalability, we will need to consider how the data is stored when designing the system, such that the performance of queries do not deteriorate noticeably as the system grows. Furthermore we should consider how we can analyse a multitude of texts per minute, ensuring that the system stays up-to-date with current trends in sentiments and topics. Finally, the architecture of the system should support the ability to scale out horizontally such as by use of a load balancer, in order to add more resources for computation power and storage as needed.

Quantifying requirements for scalable systems is a difficult and in some sense an impossible feat as it depends entirely on the context of the system. As such, we will not be able to classify our system as scalable or unscalable, but we should be able to argue in which ways the system supports scalability and in which ways it does not. Finally, we should conduct tests to see how parts of our system perform under an increasing load.

4.2.2 Robustness

As the number of concurrent users grow, our system needs to be robust, i.e. it cannot fail or slow down significantly under heavy load. There are several considerations to make here. First, since the majority of user requests will be `GET` requests fetching analysis data from the database, our system should use a Database Management System (DBMS) that allows for efficient data queries. Most writes to the database will be performed by internal services in the system, so these can be grouped into bulk transactions to reduce write time.

Furthermore, if one component of the system fails, all unrelated components and services should remain unaffected.

Finally, the system should be able to handle large amounts of traffic in a proper manner. Database read transaction requests can and should be processed in parallel.

4.2.3 Performance

Performance is a necessary non-functional requirement when building interactive systems. As mentioned in section 3.3, an unresponsive or sluggish application can lead to user disengagement. This requirement of responsiveness is directly related to the study curriculum requirement of good quality of service (see section 3.3).

We will not measure our applications performance by the response time and/or our ability to handle X request per second, as this depends on our available hardware and not necessarily our architectural design. Instead, we will evaluate the performance of our system in terms of its load tolerance. We will conduct a stress test where we simulate thousands of users performing various types of requests in order to identify any bottlenecks in our system. During the system design, we will make several design choices in order to make our system load tolerant, and the results of the stress test will be regarded as positive if the components of our system designed to be load-tolerant are not computational bottlenecks during the stress test.

We will also evaluate the performance and load tolerance of our system in terms of the computational complexity of our analytical modules. We expect the amount of data to be processed by these modules to grow very large over time. As such, the methods chosen for KE and SA should be designed and tested w.r.t. yielding good results while not becoming unusable as the amount of data grows large. In addition to simply analysing their computational complexity, a running time test should be conducted for all methods in order to decide on the best approach.

4.2.4 Usability

Usability is important for web development as an intuitive design makes a user able to achieve their goals faster and thereby have a better experience overall. Usability can be achieved by having the tools that the user is looking for visible and responsive, as well as keeping a consistent structural design across different sections of the application. Usability testing would be a good way to ensure the application is easy to use, however carries a significant time expenditure. Though usability is important it is not our focus and as such we likely wont spend a lot of time on this. We would, however, like to have at least another group of software students test our product to get at least some semi-impartial feedback at a minimal cost.

4.2.5 Testability

In order to avoid having to spend time resolving bugs later in development as well as to ensure easy maintainability, it is necessary to continuously test our application. By doing this, errors can be found and addressed early, with minimal expenditure of effort.

To do this we intend to work with TDD, writing unit-tests before implementing functionality which should ensure a wide code coverage, allowing for easy debugging. TDD is described more in section 2.2.

We will strive to ensure as much of the code is covered by unit testing as makes sense and will be trying for at least around 75% code coverage. This is not a hard requirement but more of a general guideline to see if we need to intensify our testing efforts.

To avoid artificial code coverage bloat (eg. tests that cover a lot of code but tests little to nothing), we will include our tests in code reviews.

4.2.6 Privacy

Privacy refers to keeping a user's identity and affiliated data private and secure. As we will be collecting data from users on social media, we have to take their privacy into account and handle the data in a secure way. When crawling websites, we therefore have to be aware of the data we log and how we store it.

As we are only interested in the text of posts on social media websites it may not be necessary to store an identifier for the user that wrote the post. Removing the identifier makes it harder to identify a specific user from the data we store, allowing us to focus less on securing our data. Alternatively we can hash the identifier of the user, which would allow us to mine patterns in opinions towards different brands on a user basis, while still keeping the identifier of the user pseudonymous. Hashing the identifier would still require us to look into ways of securing the data as the hash potentially could, though unlikely to, lead to the identifier getting exposed.

We have chosen to hash the identifiers as it is an obvious future expansion of the system to consider. Although highly unlikely and computationally impractical, it it possible to reverse engineer a post's author given the hashed identifier. We do not consider this a problem due to the infeasible of reverse engineering a hash by brute force, in addition to the fact that the contents of a post are removed once the processing is done, hence only a sentiment value and the associated synonym remains.

Chapter 5: Design

In this section we will describe the design of the system. First, we elicit design specific requirements from a set of synthesised user stories in section 5.1 where we also determine the vision of the system, followed by section 5.2 describing the functional requirements derived from user stories.

The design of the system architecture is described in section 5.3, followed by a description of the design of the web application and its underlying components in section 5.4. We describe the design of the data storage strategy in section 5.5, followed by a description of the user interface design in section 5.6. The design of the main analytical modules of the system, KE and SA, is described in section 5.8 and section 5.7, respectively.

5.1 User stories and vision

In this project, we are not building a system for a specific client. Instead, we are building a system in order to solve an existing problem within user opinion mining, as stated in section 3.4. Since the system has no concrete end-user at this point in time, the following presents a series of synthesised user stories that serve as an outline for the design and functionality of the system. In the following user stories, our system will be referred to as SentiCloud.

5.1.1 User creation and setup

John Doe runs a software company, Geegle Inc., that provides a search engine over the web for free. John aims to make money from targeted advertisement based on user search histories and interests.

John has a hard time figuring out what the users of his system think about the quality of the advertisements. Additionally, as the field is saturated by different search engines, John is interested in seeing where he lies w.r.t. user opinion compared to his competitors. To gather this information, John will use SentiCloud.

John enters the SentiCloud website. He is prompted to create or log in with an existing account. John has no account, so he creates one by entering a username, an e-mail and a password.

After creating an account, John is logged in and is presented with the brands he is currently monitoring in a table. John sees that he is currently not monitoring any brands, so he wants to create one.

5.1.2 Creating a brand with synonyms

John presses the plus button on the brands page and is prompted by a dialog with a brand creation form. John is asked to enter the name of the Brand he wants to monitor. John wants to monitor his own company, so he enters "Geegle" and presses the button that says create.

The dialog closes and the brand now shows up in the list of brands John is monitoring. He clicks the brand name and is taken to a page with a list containing synonyms with "Geegle" as its only entry. John hovers over a question mark next to the box, after which a tool-tip explains that the synonyms serve as the basis of information search about his brand. Currently, John will only see user opinions from SMPs regarding Geegle, if Geegle is commonly referred to in another way he should add some synonyms for the brand.

John enters "search engine", "fake Google" and "AddSense" into the input field and presses enter in between each to add them to his list of synonyms for Geegle. Finally, John is informed the system will start gathering data and to check back later.

5.1.3 Analysing user opinion

After creating the brand Geegle and waiting for a couple of weeks, John visits SentiCloud once again. John expects to see at least a couple of weeks worth of user opinions about his brand. John logs in with his SentiCloud account and is presented with the table of his brands. He sees that the columns sentiment, sentiment trend and posts columns have been updated with data for the given day. He clicks "Geegle" to see more details about his brand.

John is taken to the details page for "Geegle" and sees that a graph has been added to the details page depicting the user opinions regarding "Geegle" and its synonyms. One of the graphs is a two-dimensional coordinate system with a number of differently coloured lines across it. From the legend in the graph, John can see that each line represents a synonym of his brand. John can also see that the y-axis of the graph represents the overall sentiment towards his brand which splits at the middle of the y-axis: above the middle shows positive sentiment, the middle shows indifferent sentiment, and below the middle shows negative sentiment towards his brand. The x-axis represents time, and defaults to show a week worth of data up until today with a data point for each day in the week.

John hovers over one of the points on one of the lines and is presented with a list of five positive and five negative words. John reasons that these words represent what the majority of the users are saying in SMPs regarding the given synonym at a given point in time.

In the right corner of the screen is a button with a cogwheel icon, John reasons that this must be the settings for the details page. He clicks it and a settings bar appears that allows him to change the start and end date for the depicted data as well as the granularity. John changes the starting date to a few days prior to the default starting date and clicks the "Update" button. The graph disappears and a spinning wheel takes its place. After a short while, the graph is redrawn with the new data and returns to the view removing the spinning wheel.

5.2 Functional requirements from user stories

We are able to determine some more precise functional requirements from the presented user stories. The following presents specific functional requirements for the system.

5.2.1 Authorisation functionality

The system should support user authorisation, as such, we need to store account information such as email and passwords in a secure and reliable way.

5.2.2 Brand creation and synonym monitoring

Every user has the ability to create their own brand for monitoring with a series of synonyms. Furthermore, the system should monitor brand synonyms on a daily scheduled basis. The system should therefore store information regarding which users are monitoring which brands. Here, it is worth noting that the monitoring results of a brand is that of its underlying synonyms. As such, the system should gather information regarding all existing synonyms on a scheduled basis, and then be able to group the analysis results to users who have selected them as part of their monitored brands.

5.2.3 Data analytics

The system should be able to present monitoring results to the user as a two-dimensional graph. The y-axis should represent the overall sentiment towards a synonym, the x-axis should represent the time, and when hovering over a line in the graph, the user should be presented with the top five positive and negative keywords and phrases for that synonym at the point in time. This requires the system to store the sentiment, keywords and time for every synonym on a daily basis.

5.3 System design

This section presents the design of the system architecture and underlying components. The decisions made while designing the system were made in accordance with the functional and non-functional requirements stated in section 4.1 and section 4.2. The following design of the system architecture and individual components represents the base system to be built to allow SA and KE to be conducted.

In subsection 5.3.1 we will describe the overall architecture and why we choose this structure, followed by subsection 5.3.6, describing the flow of our system. We will describe the different components of our architecture in subsection 5.3.2, subsection 5.3.3, subsection 5.3.4, and subsection 5.3.5.

5.3.1 System architecture

The system architecture can be seen in Figure 5.1. The following describes how the requirements in chapter 4 have been handled in the architecture design.

Robustness and performance

In order to comply with the requirement of robustness (see subsection 4.2.2), all functional components of the system are implemented as self-contained services. As such, if one service fails, it will not affect any other service apart from failing to respond to incoming requests. When a service fails, it can be easily re-deployed.

The architecture is built after the gateway routing design pattern. This can be seen in Figure 5.1 in the `Gateway` component handling routing incoming requests to their appropriate services and aggregating the resulting response before sending it to the client.

Client requests are first processed by a load balancing service that prompts a series of separate servers for their availability and current load. This prompt is done through the gateway component on each server. When a server with good availability has been chosen, the request is routed to that server's gateway that subsequently processes the request appropriately.

The following sections describe the purpose and structure of the components and their underlying services.

5.3.2 Gateway component

The gateway component, as mentioned in subsection 5.3.1, is responsible for routing client requests to the appropriate services. Composed of its underlying services (`Authorization`, `Brands`, `Accounts`, `Synonyms` and `Statistics`), the gateway component stores all its data in its assigned DBMS. All services in the gateway component communicate with the gateway DBMS through the common Object-Relational Mapping (ORM).

A detailed representation of the component can be seen in Figure 5.2. All public methods, denoted with a +, denote that the method is a publicly available API endpoint.

Authorization service

The `Authorization` will be invoked for all incoming client requests in order to verify whether the client is coming from a logged in and valid user. The service is also responsible for validating user logins.

Accounts service

The `Accounts` service is responsible for creating and deleting user accounts from the system.

Brands service

The `Brands` service is responsible for creating, deleting, and fetching brands for a specific account.

Synonyms service

The `Synonyms` service is responsible for creating, deleting, and fetching synonyms. It also provides a private endpoint, `get_new_synonyms`, that is called by the scheduler service in the analysis

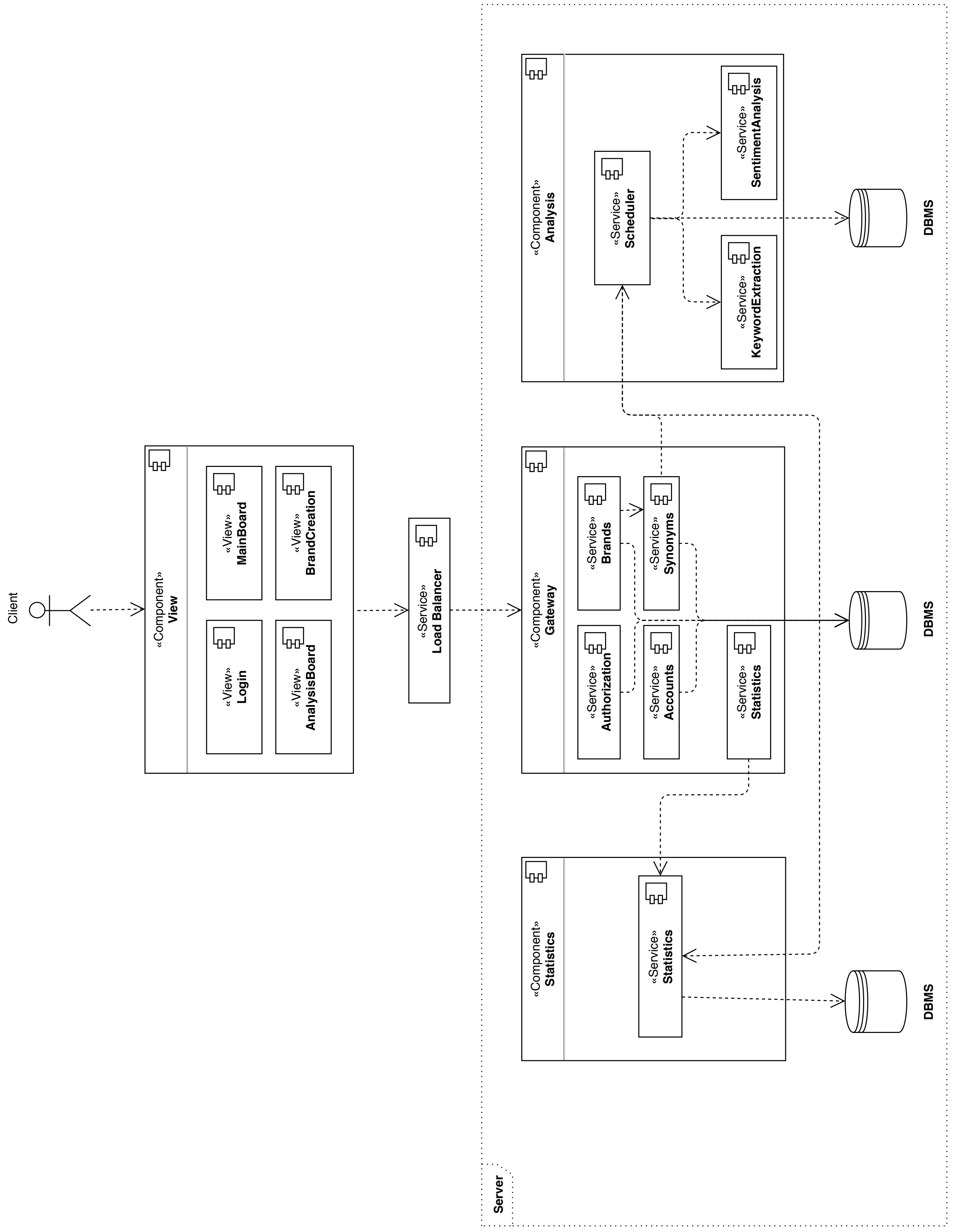


Figure 5.1: The overall architecture of the system. Dotted arrows represent dependencies between components and services.

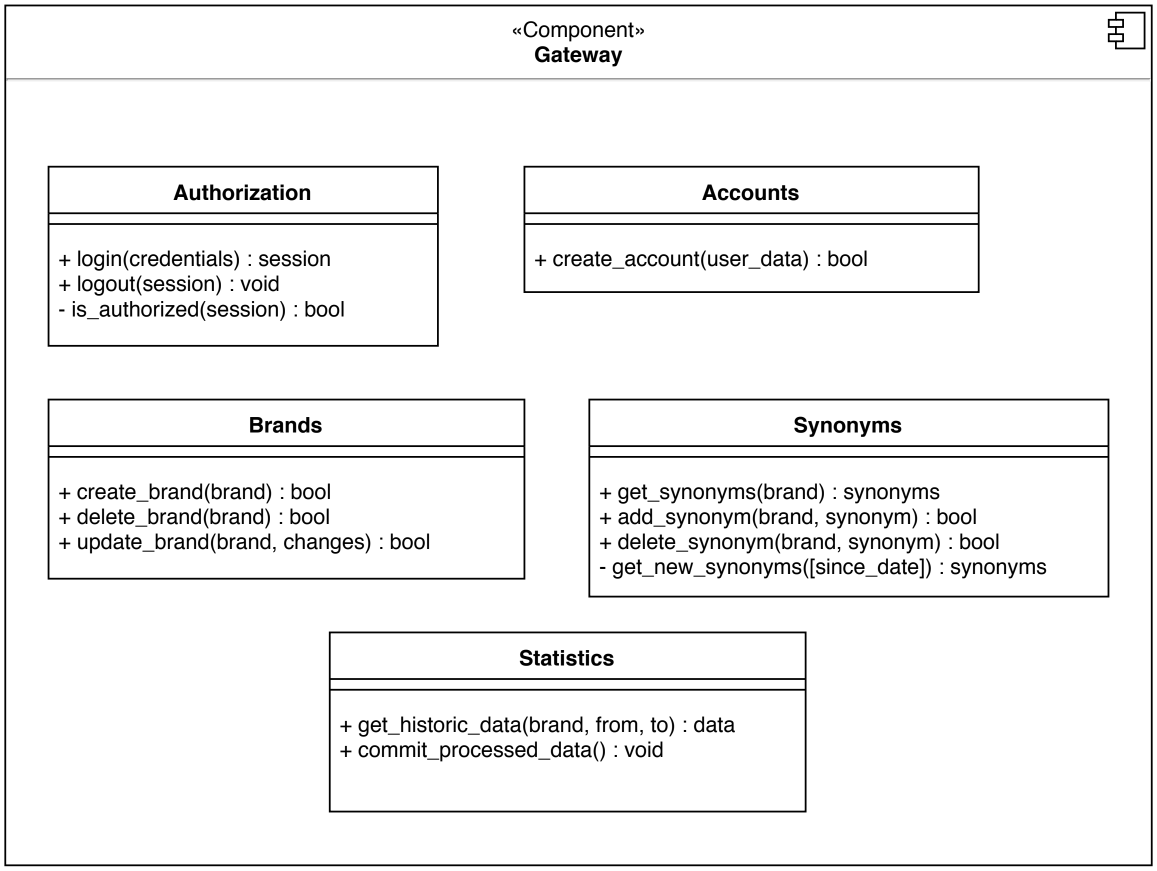

«Component»
Gateway
Authorization
+ login(credentials) : session
+ logout(session) : void
- is_authorized(session) : bool
Accounts
+ create_account(user_data) : bool
Brands
+ create_brand(brand) : bool
+ delete_brand(brand) : bool
+ update_brand(brand, changes) : bool
Synonyms
+ get_synonyms(brand) : synonyms
+ add_synonym(brand, synonym) : bool
+ delete_synonym(brand, synonym) : bool
- get_new_synonyms([since_date]) : synonyms
Statistics
+ get_historic_data(brand, from, to) : data
+ commit_processed_data() : void

Figure 5.2: Detailed description of the gateway component services. Publicly available API endpoints are represented with a +, and are otherwise represented with a -.

component when new synonyms are to be scheduled for crawling.

Statistics service

The statistics service is responsible for handling data statistics requests. This service is the only service in the gateway component that has a direct connection to the Statistics component. When a client request queries for statistical data over some brands, the Statistics service fetches related synonyms from the Synonyms service. It then uses the Statistics service in the Statistics component to fetch the historical data which is then returned to the user.

5.3.3 Analysis component

The analysis component is responsible for data gathering and processing. It contains three services: the Scheduler, the KeywordExtraction, and the SentimentAnalysis service. A detailed representation of the component services can be seen in Figure 5.3.

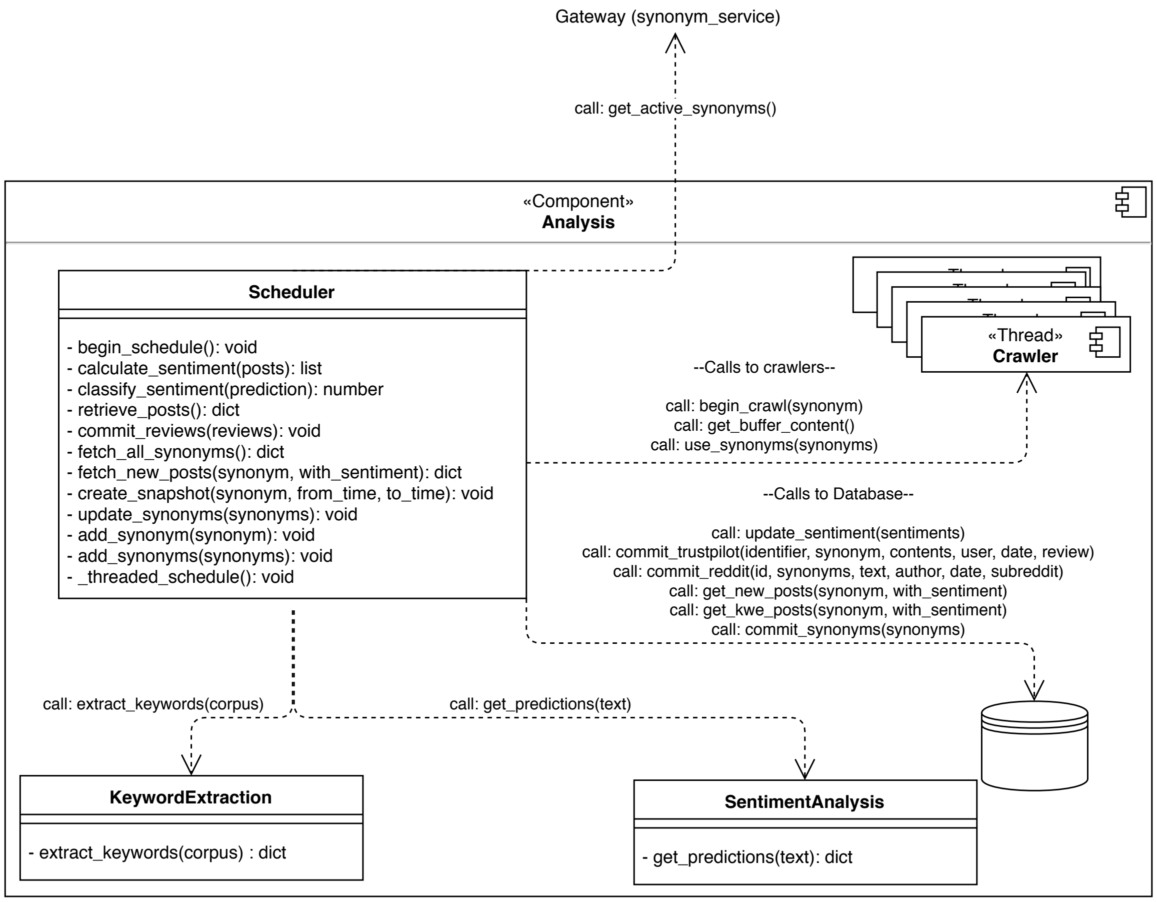

Gateway (synonym_service)
call: get_active_synonyms()
«Component»
Analysis
Scheduler
- begin_schedule(): void
- calculate_sentiment(posts): list
- classify_sentiment(prediction): number
- retrieve_posts(): dict
- commit_reviews(reviews): void
- fetch_all_synonyms(): dict
- fetch_new_posts(synonym, with_sentiment): dict
- create_snapshot(synonym, from_time, to_time): void
- update_synonyms(synonyms): void
- add_synonym(synonym): void
- add_synonyms(synonyms): void
- _threaded_schedule(): void
«Thread»
Crawler
--Calls to crawlers--
call: begin_crawl(synonym)
call: get_buffer_content()
call: use_synonyms(synonyms)
--Calls to Database--
call: update_sentiment(sentiments)
call: commit_trustpilot(identifier, synonym, contents, user, date, review)
call: commit_reddit(id, synonyms, text, author, date, subreddit)
call: get_new_posts(synonym, with_sentiment)
call: get_kwe_posts(synonym, with_sentiment)
call: commit_synonyms(synonyms)
call: extract_keywords(corpus)
call: get_predictions(text)
KeywordExtraction
- extract_keywords(corpus) : dict
SentimentAnalysis
- get_predictions(text): dict

Figure 5.3: Detailed representation of the analysis component services.

Scheduler service

As the name suggests, the scheduler is responsible for scheduling information gathering regarding the existing synonyms in the system. Whenever the scheduler decides to begin information gathering for a set of new synonyms (which are fetched from the private endpoint `get_active_synonyms()` in the synonyms service on the gateway API), it adds the synonyms to the search set used by the crawlers running in separate threads.

The crawlers will hand back the data they have gathered along their crawl in intervals. This textual data is stored in a database assigned to the scheduler. The sentiment of the posts is calculated immediately using the `SentimentAnalysis` service before committing them to stable storage.

When the scheduler decides that it is time for processing synonym data, it will query the local database for saved textual data regarding those synonyms. It will then use the `KeywordExtraction` service to determine what users of similar sentiment are saying about a synonym.

When the processing has finished, the scheduler will use the function `create_snapshot(synonym, from_time, to_time)` which will save the sentiment in storage for the system to use for tasks such as statistics.

5.3.4 Crawler

The crawler seen in Figure 5.3 is implemented as a simple focused crawler that only fetches data from the Trustpilot website. Trustpilot is a website on which users can review the services of an official entity (e.g. a company). Since the crawler is only fetching data from a single host, its design and implementation is simple. The frontier is implemented as a simple queue of synonyms to crawl. When a synonym is popped from the frontier, the crawler performs a search on Trustpilot in order to fetch review pages that relate to this synonym. The structural design of the crawler can be seen in Figure 5.4.

Frontier

As can be seen in Figure 5.4, the frontier is a doubly-layered queue structure. When the crawler pops a synonym from the frontier, a new queue, `SynonymURLQueue`, is received. From this queue, a Uniform Resource Locator (URL) is popped an subsequently crawled for user reviews. From the user reviews, the text and date is extracted, cleaned, and stored in a temporary database. Links pointing to subsequent user review pages are enqueued in the `SynonymURLQueue` for later retrieval. When a `SynonymURLQueue` has been processed, it is enqueued into the frontier queue again, making sure that the crawler will process the same synonym at a later point in time where new reviews may have arrived.

Ensuring fresh content

Trustpilot reviews are displayed chronologically w.r.t. when they were posted, and carry with them the username of the user who posted the review. When a page is received by the crawler, it processes the first review and determines the date and username. The crawler maintains a dictionary with dates as keys, and a list of hashed usernames as the value. In this way, the crawler knows, for every date,

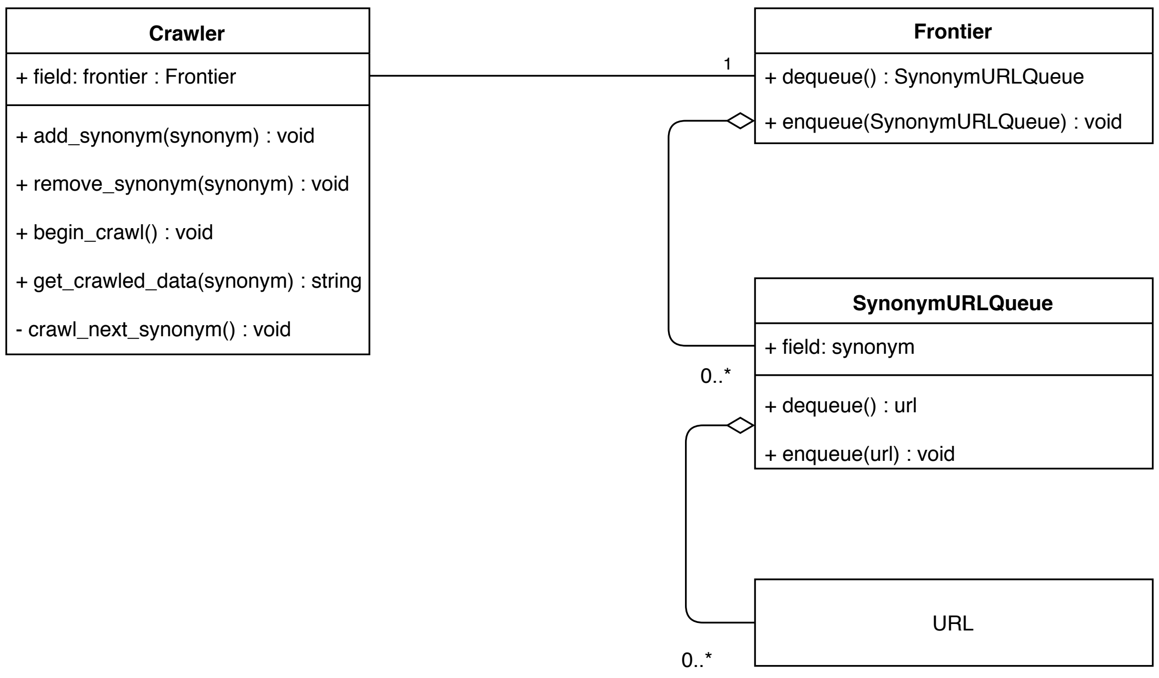

Crawler
+ field: frontier : Frontier
+ add_synonym(synonym) : void
+ remove_synonym(synonym) : void
+ begin_crawl() : void
+ get_crawled_data(synonym) : string
- crawl_next_synonym() : void
1
Frontier
+ dequeue() : SynonymURLQueue
+ enqueue(SynonymURLQueue) : void
SynonymURLQueue
+ field: synonym
+ dequeue() : url
+ enqueue(url) : void
0..*
URL
0..*

Figure 5.4: Structural design of the Trustpilot crawler.

what users have posted a review. If the date and username already exists in the dictionary, the review is discarded and the review page is not processed further. The `SynonymURLQueue` is enqueued into the frontier queue again.

Politeness

Trustpilot has given us permission to crawl review data from their website. As such, since we are only fetching links directly relating to review pages for a given entity, we are not checking the `robots.txt` file for allowed URLs. Furthermore, since the crawler is only fetching data from one host, the crawler simply ensures a 2 second wait time between fetching URLs from the frontier in order to ensure politeness.

5.3.5 Statistics component

The statistics component is responsible for providing user queries with statistical data regarding a brand or collection of synonyms and their corresponding sentiment and keywords. The statistics component contains a `Statistics` service that is used by the `Statistics` service in the `Gateway` component to fetch data for user queries. It also provides an endpoint used by the `Scheduler` service to store newly extracted keywords and sentiment.

It is worth noting that the endpoint in the gateway component's `Statistics` service, `get_historic_data(brand, from, to)` is not the same as the `get_statistics(synonym, from, to)` endpoint in this component. When a user queries for a set of historical data regarding a brand, the gateway component uses the `Synonyms` service in the gateway component to retrieve all synonyms related to that brand. The

Peter - text runs
border

service in the gateway component then uses the `get_statistics(synonym, from, to)` endpoint in this component in order to receive data regarding all related synonyms. The data is then aggregated according to the query and returned to the user.

A detailed description of the `Statistics` component and its service can be seen in Figure 5.5.

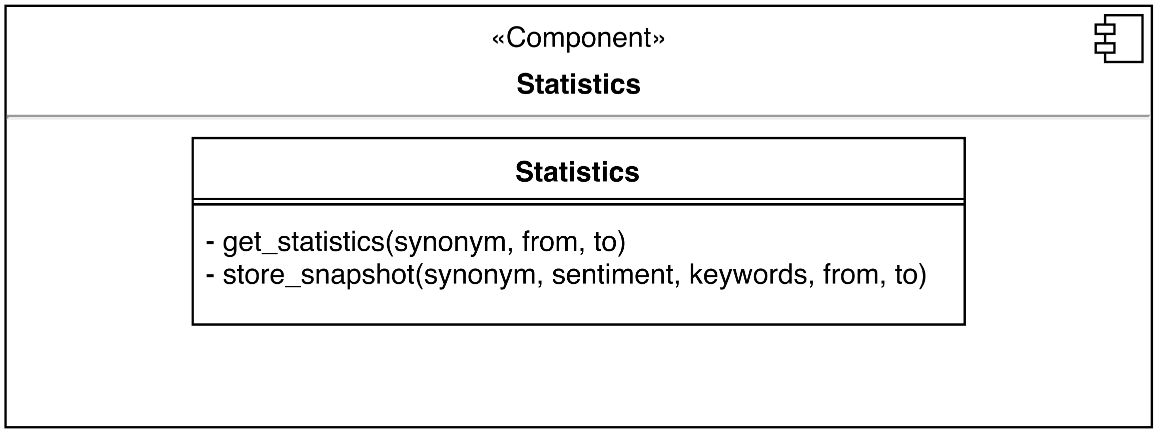

«Component»
Statistics
Statistics
- get_statistics(synonym, from, to)
- store_snapshot(synonym, sentiment, keywords, from, to)

Figure 5.5: Detailed representation of the statistics component and its `Statistics` service.

5.3.6 System flow

In this section we will go over the system flow and describe the flow of interactions. The basic flow will be described in text and will be based on Figure 5.1. We will describe the flow of the scheduler in detail as it takes multiple steps to process our data.

An interaction with the User Interface (UI) that requires data, such as a query for a brand over a time range, goes through the gateway component. The gateway component is responsible for authentication and user data storage and acts as a mediator between the UI and the statistics component.

The scheduler component is a self contained service that will run independently of the remaining components. It is responsible for scheduling when data should be retrieved, updated and combined into a snapshot.

When data is due for retrieval, the scheduler queries the crawlers for data and commits it to stable storage. When data is due for updating, the scheduler fetches data from stable storage, sends it to the SA component to be classified and updates the stable storage with the classified data. When a snapshot is due, the scheduler fetches classified data from stable storage, created within a given time span denoted by the snapshot specifications, sends the data to the KE component and bundles it up before sending it to the statistics component.

The statistics component acts as a data access layer. Upon receiving data from the scheduler component, the statistics component commits the data to stable storage and upon receiving a data request from the gateway component, the statistics component will fetch and return the data from stable storage.

5.4 Web application

In this section we will describe the design decisions we have made with regards to the web application. This entails various aspects including the overall design as well as the concrete choices in choosing the technology stack. For all the choices we have made, we will justify the choice in relation to both the functional and non-functional requirements defined in chapter 4.

5.4.1 Render side

An important decision to make for a web application is deciding which party in the client-server relationship is responsible for building the markup document used to render the different pages or components of the application. When using Server-Side Rendering (SSR), the server parses requests sent by the client and then returns entire markup documents (typically HTML) which is then rendered in the client's browser. On the other hand with Client-Side Rendering (CSR) the client only uses the server as a source for data, typically serialised in a format like JSON. Based on the data it receives, the client then constructs markup documents which are rendered in the browser. This is not a binary situation, however, as some applications employ a hybrid method which makes use of both.

Bliver HTML file
dered" i browsere

None of our non-functional requirements strongly favour any of these approaches, but there are arguments as to why CSR is favourable in the case of our system. First of all, the cleaner separation of view logic and data logic arguably leads to easier testability. A cleaner separation also means that implementing the platform as a mobile application would not require much work beyond the UI design. Furthermore, CSRs may give the user a notion of increased performance since the entire page does not have to reload whenever an action is made.

CSRs have notable issues which should be considered. The complete separation of view and logic means that the client will include models and other domain knowledge which is already present on the server. This will not only result in duplicate code, but also require more development time, especially when some of these models are inevitably changed. Furthermore, using CSR means that web-crawlers will encounter whichever script is responsible for rendering and not the dynamic content being served. This in turn makes it unfeasible for a search engine to index the website. However, this is not a concern for us since we are developing a dashboard which will be inaccessible for unauthorised users.

A hybrid method is also a possibility. Such a method provides some of the responsiveness provided by CSR, while still allowing web-crawlers to crawl the website. For platforms like social media, such an approach makes sense, given that user-generated content should be indexed by search engines, while still being able to provide the flexibility and responsiveness of CSR. Notably the system will become increasingly complex with such an approach because controllers on the server will need to accommodate both markup templates and serialised data. We will not delve further into this method since, as previously stated, we do not want to expose our application to web-crawlers.

Based on our review of the different methods, we believe that CSR is the best fit for our system. As an implication of that choice, it follows that we need some API to provide the client with serialised data. To accommodate this need, we will be implementing a single RESTful API. A notable property about RESTful APIs is that they provide a lot of flexibility in having a stateless design. Being stateless, no details are retained between calls. This effectively means that the components of the

Er det ment som
EN api kommuni
med client? Eller
systemet består af
api?

API can be replaced if they fail under load, which fits nicely with our non-functional requirement of scalability. Furthermore, as illustrated in Figure 5.6 the API can be used by than a single application. This is easily achieved because a RESTful API is client agnostic, as long as the client conforms to the model representations governed by the API. This fits well with one our of functional requirements, which states that separate client systems should be able to use the system.

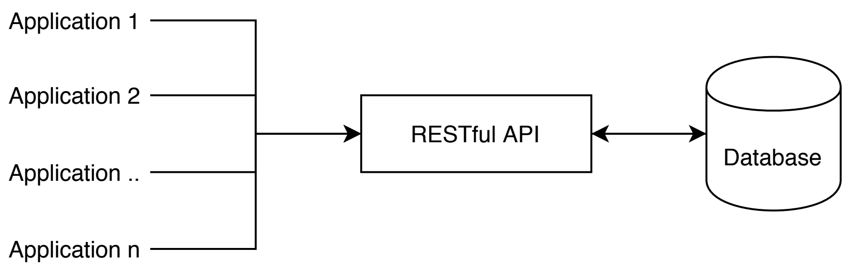


Figure 5.6: Several application using the same API.

ure this is cor-

5.4.2 Microservice architecture

This section will cover what the microservice architecture is, its benefits and drawbacks and how these relate to our project.

The microservice architecture tries to keep components loosely coupled while still allowing collaboration between them [44]. Each service should handle a specific area and only have functions related to it. The services communicate using synchronous or asynchronous protocols, as HTTP/REST and AMQP respectively. There are no common databases. Instead each service has its own database if needed. There therefore needs to be data consistency across the databases.

The Saga architecture is implemented to handle consistency across databases, by having events that ensures the consistency and compensating activities if the update of one databases is not allowed in another [44]. It is not necessary to have distributed transactions as consistency is handled elsewhere. However, though the Saga architecture increases complexity of our program, as compensation activities have to be implemented that explicitly undo previous changes in a saga in case of an error, it still increases decoupling.

As described, services use protocols for communication rather than calling methods directly from another service. An often used method in the microservice architecture is the *smart endpoints and dumb pipes* [29]. The most common protocols used are HTTP request-response and lightweight messaging. Here the end point is a service and the pipes either simple HTTP requests or a message router. The services are the smart endpoints and are message producing and consuming. Furthermore, the application should be coarse-grained, meaning fewer, larger services, as a fine-grained program would lead to chatty communication between services and would not perform well because of the communication pattern.

Since each service runs independent from each other it is necessary for the application to be able to handle a failure of a service [29]. The application should be able to detect a failed service and possibly be able to restore it as well. Extensive monitoring and logging setups are often required to analyse and fix bad emerging behaviour.

Using microservices enables us to develop multiple parts of the program concurrently and scale the service. The development is slowed in the early stages because of the increased complexity, but the microservice architecture enables better scaling capabilities as it reduces tangled dependencies and has a natural functional decomposition of the program [44]. It also enables us to use different tools for specific aspects of the application, i.e. performance critical aspects can be written with languages that support this requirement. We will not be implementing all the monitoring aspects of the microservice architecture, as it is outside our focus, but it should be implemented if the program should be deployed.

5.4.3 Service oriented architecture

The microservice architecture is a subset of the service oriented architecture and as such the two are quite similar while still having their differences.

Similarly to the microservice architecture described in subsection 5.4.2 the service oriented architecture also tries to keep components loosely coupled while still allowing collaboration between them. The service oriented architecture is not as strict as the microservice architecture. For example the requirement from the microservice architecture which states that each service should have its own database is not a requirement in the service oriented architecture. This means that one database service can be used amongst any combination of the remaining services in the system in the service oriented architecture.

The service oriented architecture typically has a central component which acts as the link between the users of the system and the system itself.

While the microservice architecture promises more independent services that are guaranteed to be able to be deployed independently it is not always the best solution as it also comes with increased redundant information that may be unnecessary, which is a problem that is not as pungent in the service oriented architecture [35].

5.4.4 Front-end application frameworks

There are two main design patterns for web applications today, namely Single-Page Application (SPA) and Multiple-Page Application (MPA). A SPA does not require page reloading. As the name suggests, it is a single web page that you visit which then loads all other content using JavaScript. A MPA on the other hand requires page reloading.

The initial loading time of a SPA is a fair bit longer than the loading time of a MPA, but once the application is loaded, a SPA is more responsive. Therefore SPAs are often preferred over MPAs when building web applications where users tend to hang around for a while and browse, e.g. Facebook, Twitter and Gmail, as opposed to quickly moving on to a different web application.

Because we plan to construct an analytics platform where the user can log in and interact with our model, we expect the users to hang around for a while and thus it makes sense to limit this subsection to exploring the various Web Application Frameworks (WAFs) designed specifically for building SPAs.

React

React is an open-source JavaScript library that can be used as a base in the development of SPA. React is not a full fledged WAF as the development of complex SPA usually requires the use of additional libraries for state management, routing and interaction with an API.

React is a very focused solution, it focuses only on the view part of the Model-View-Controller (MVC) architectural pattern, making it very lightweight and thus allowing fast render times. The big community of React is constantly creating useful packages that can be used to extend the functionality of or work in tandem with React. This, however, could introduce support issues in the future.

Vue

Vue is an open-source JavaScript framework designed for building UIs, but can also function as a WAF for building complex SPA.

The advantage of Vue, in comparison to the aforementioned WAFs, is its small size and ease of use, despite having a large number of built-in components. Unit testing is also fairly easy, as any test runner compatible with a module-based build system works.

The disadvantage of using Vue is its small community and therefore lack of packages, which is a result of being relatively new. This means that a lot of packages readily available to other WAFs has to be implemented manually or alternatives have to be found of which there are few.

Angular

Angular is an open-source JavaScript WAF designed for building complex SPAs, using the MVC architectural pattern.

The advantage of Angular compared to the previously discussed libraries is the fact that Angular includes all the necessary building blocks in its base implementation and thus does not require the inclusion of third party packages to extend functionality, alleviating potential support issues. Additionally Angular includes the JavaScript test runner Karma for unit tests and its own end-to-end test framework Protractor.

The disadvantage of Angular is the complexity. Angular is not simple to use, it has a steep learning curve, because of the size of the WAF and the limited backwards compatibility, which is reflected in the long API documentation.

Choice of front-end framework

To determine which WAFs is suitable for our application, we look back at our non-functional requirements described in section 4.2. As data and authorisation privileges is handled by a controller separate from the front-end framework as shown in Figure 5.1, privacy is not an issue in choosing the front-end framework. Additionally the three WAFs should provide the same possibilities in terms of UI design and thus usability is not an issue either. That leaves us with three non-functional requirements to consider: performance, scalability and testability.

In terms of performance, all three WAFs perform similarly. They all compile the written code into code that's highly optimised for modern JavaScript virtual machines [43]. Several online benchmarks exist and they generally agree that there are insignificant differences between the described libraries in measures such as the time to get the first meaningful render.

In terms of scalability, it is necessary to utilise a modular build system to organise code in large projects. Angular comes with such build system, which allows the construction of modules consisting of related components along with their dependencies, which can be developed separately and compiled into the final application. Vue and React on the other hand do not come with a modular build system, but there exists a variety of third party modular build systems such as Webpack or Browserify, that can be used instead. Therefore it can be argued that all three WAFs are scalable, but we believe the structure of Angular leads to a more natural development of modular systems [13].

Lastly, in terms of testability, Angular comes with a built in JavaScript test runner for unit tests as well as an end-to-end test framework, whereas Vue and React require the inclusion of third party test runners. The fact that the test runners are included in Angular ensures continuous support of the test runners, whereas the chosen third party test runner for Vue and React will depend on the fact that the test runner will be maintained. However, that is more of a maintainability concern than testability concern and thus disregarding the maintainability aspect all three WAFs fulfil the testability criteria.

In addition to the factors mentioned above, some of the group members have experience in developing Angular applications, which reduces the adoption time and facilitates information sharing within the group. While we have not listed it as one our non-functional requirements, it is obvious that another benchmark should focus on the developer productiveness and in that case, we believe that Angular would be the best fit. As such, we choose to use Angular as our front-end framework. To summarise our reasoning, Table 5.1 provides a useful reference.

	Angular	React	Vue
Performance	Comparable, depends on use case		
Scalability	**High, modular design encouraged**	High	
Testability	**High, built-in test runner**	High, external test runner	
Learning curve	Steep	Intermediate	**Gentle**
Team experience	**High**	Low	None

Table 5.1: Summary of WAF comparison.

5.4.5 Back-end application technologies

The client-side documents presented through the front-end frameworks described in subsection 5.4.4 must be served by a back-end application typically referred to as a server. The server is responsible for processing the data and sending the results to the front-end, so it is crucial that the server is implemented using efficient technologies and in an easily scalable fashion.

Python

Python is a dynamically typed and interpreted language for which there exists comprehensive web application- and ML frameworks (e.g. Django[1] and TensorFlow[2], respectively) with which we have experience using. As such, Python can be used to implement our entire back-end, including data pre-processing, construction and training of ML models, and the APIs used by the front-end.

However, Python is known to be slower than most strongly typed languages for several reasons (e.g. dynamic typing and its implementation of lists and dictionaries) [41] which could become a problem when the amount of data to be processed becomes too large, e.g. during training of an intelligent model or during graph building for KE. Fortunately, several Python libraries for working with machine learning exist and are implemented as a Python wrapper around a C++ back-end (e.g. Graph Tool[3], TensorFlow[4], and Keras[5]), allowing us to benefit from the readability and writability of Python as well as the performance of C++.

C++/C

C++ and C perform extremely well compared to most other strongly typed and compiled languages [41]. High performance web application frameworks such as CppCMS also exist for the languages[6]. While ML frameworks libraries exist for the languages, as well as for TensorFlow with which we have experience, the support and documentation is limited compared to other languages e.g. Python, Java, and C#. Lastly, none of the group members are experienced in writing large systems using C++ and C, both of which have a notoriously steep learning curve. It is important to consider this as it could slow down our implementation phase significantly.

C# and .NET Core 2.1

C# is a strongly typed, compiled language with which all group members are experienced. Furthermore, there exist both web application-[7] and ML[8] frameworks. While it does not perform as well as C++ and C, it performs better than most dynamically typed and interpreted languages such as Python [41]. However, .NET applications run on a virtual machine with a garbage collector, introducing some inherent performance overhead. The web application framework ASP.NET Core 2.1 has built-in enforcement of the MVC architectural pattern, which can help keeping a proper separation of components and responsibility across the system. Furthermore, an interesting advantage of the ML framework ML.NET is that it supports the use of TensorFlow models that have been trained using any implementation of the TensorFlow framework, e.g. Python [4].

[1]Django (`https://www.djangoproject.com/`)
[2]TensorFlow(`https://www.tensorflow.org/`)
[3]`https://graph-tool.skewed.de/`
[4]`https://www.tensorflow.org/`
[5]`https://keras.io/`
[6]CppCMS (`http://cppcms.com`)
[7]ASP.NET Core 2.1 (`https://docs.microsoft.com/en-us/aspnet/core/`)
[8]ML.NET (`https://www.microsoft.com/net/apps/machinelearning-ai/ml-dotnet`)

Choice of primary language

We have decided to implement the entire back-end of our system with Python as the implementation language. Python will be used for building the server, services, ML models and KE graphs. Most of the heavy data processing will be done using frameworks and libraries with a C++ back-end (e.g. TensorFlow for ML and Graph Tool for graph building). According to language comparison benchmarks [41], C++ applications run with an acceptable performance and efficiency.

Furthermore, threading libraries exist for Python allowing us to further take advantage of multi-threading between processes. This will allow the system to scale well w.r.t. performance as the amount of data grows. Furthermore, building the entire system using only one language will likely help avoid complications during development and maintenance. Following these decisions, we will be fulfilling our non-functional requirements in section 4.2.

For building the web application, several Python web development frameworks exists, e.g. Django and Flask. Since the server built with these frameworks will be running as a Python process, other Python processes like ML sessions in TensorFlow and KE graphs can be easily integrated into the running server processes, fulfilling our functional requirements stated in section 4.1.

	Python	C#	C++
Performance	Slow	Fast	**Very fast**
Testability	**High, libraries are available**		
ML tool support	**Good**	Limited	
Learning curve	**N/A***		Steep

Table 5.2: Tabular summary of programming language comparison. * Since we already are familiar with these languages.

In Table 5.2 we have made comparisons between our discussed languages that could be beneficial for the server setup. Ultimately all three languages would perform well enough for our solution with C++ being the best performance wise. However, C++ comes with more downsides than the remaining two, from our perspective, as none of us have much experience with it and it has a steep learning curve which would slow us down a lot.

We are choosing Python over C# as the tools we plan to use, mainly TensorFlow, have greater support for Python compared to C# which might save us some hassles and thereby allow less wasted time. Python also has a good write- and readability which we deem beneficial enough to us to compensate for the worse performance compared to C#.

5.4.6 Web application frameworks

To facilitate our web application development we will be using a framework. We will look at and compare several different python frameworks, with a focus on speed. That is, we will mainly look at how quickly we can develop an application using each framework as well as how fast the performance will be. Furthermore, we will be using frameworks for supporting the development of our analytical modules, i.e. SA and KE.

Django

Django[9] is a framework that follows a "batteries included" principle, that is most functionality is provided without the need for extra libraries.

By using an architecture where each part is independent, and thus easily replaceable or changeable, Django is scalable as extra hardware can be added at any bottleneck that might arise without unduly influencing the rest of the system.

Flask

Flask[10] is a micro framework with a minimalist approach. This means that unlike Django where most functionality is part of the base framework, Flask only provides a very limited base functionality. If you need other functionality you then need to add additional libraries, of which there is an extensive collection. Flask generally performs comparably or faster than Django [42].

Falcon

Like Flask, Falcon[11] is a microframework and thus provides only a limited basic functionality. However, Falcon is designed specifically to build RESTful APIs. Like with Flask, additional functionality can be achieved with the use of extensions but as a younger framework the selection of available libraries for Falcon is more limited than for Flask. Of the three frameworks we have looked at Falcon is generally the fastest [42].

Choice of web-development framework

	Django	Flask	Falcon
Performance	Slowest	Medium	**Fastest**
Native Functionality	High	Low	Very Low
Selection of Extensions	**Good**		Limited

Table 5.3: Tabular summary of web framework comparison.

As a fully fledged framework, Django offers extensive functionality, however, much of this functionality will be unnecessary for any given component. With a micro framework we will be able to tailor the functionality to each component with the addition of the necessary extensions. While Falcon may offer somewhat faster performance than Flask, it is perhaps too bare-bones with functionality that most components need missing. This combined with the lower availability of extensions, have led us to choose Flask as our web framework.

Next, we will choose between the frameworks that we will be using for the implementation of our analytical modules.

[9]Django (`https://www.djangoproject.com/`)
[10]Flask (`http://flask.pocoo.org/`)
[11]Falcon (`https://falconframework.org/`)

5.4.7 Graph building framework

Part of our solution relies on being able to extract keywords from a corpus using the Keyword Graph (KG) approach described in subsection 5.8.2 using graphs. This approach encompasses several different settings that all vary in how the graph is built and queried. To ease this process we will be using a framework to handle these tasks. As mentioned in subsection 5.4.5, Graph Tools is a graph building library for Python with a C++ back-end, making it quite fast. Another alternative to graph building is NetworkX[12], a graph building library implemented in Python. This library, while running at slower speeds than that of Graph Tool, makes good use of Python's dynamic typing and is very easy to use.

In order to reduce the development time w.r.t. building and querying graphs, we will be using NetworkX. While the NetworkX framework is expected to run slower than Graph Tool, we regard reduced development time as more valuable than the performance of the frameworks. Our solution needs to be scalable, and if our calculations and data are not designed in a properly scalable way, the performance of the implementation language will not matter as the amount of data grows larger.

5.4.8 Machine learning framework

For building our SA module, we will be using an ML framework. The group members are experienced with both the TensorFlow and Keras frameworks, which as mentioned in subsection 5.4.5 use a C++ back-end. For this project we have chosen Keras as it is an abstraction over TensorFlow (and other libraries), allowing us to more easily implement and train the models that we will be using for SA. Since the frameworks run on the same C++ back-end, we do not expect to lose any significant performance by choosing Keras over TensorFlow.

5.4.9 System architecture

The overall system architecture and general flow of interaction between the server and the client can be seen in Figure 5.1. The following will describe how the different parts of the interaction chain will be implemented.

At the start of a new client session, the client makes a request for the system web application. The Python server responds with an Angular SPA. This interaction (represented by the red lines in Figure 5.1) only happens once at the start of every client session.

Once the SPA has been loaded onto the client system, the client can make several subsequent calls to APIs provided by the Python server (represented by the green lines in Figure 5.1). These calls can be requests for another page within the SPA, initialising SA, KE, and more. Once the server receives an API request, it will route the request to the appropriate microservices running as independent processes in Docker containers on the server side. The microservices handle the requests and respond accordingly. The server aggregates the result of each responding microservice and returns the result to the client.

[12] `https://networkx.github.io/`

5.4.10 API documentation

As we have previously mentioned, the backend part of the web application will consist of a RESTful API which will communicate with our services, which is also by means of RESTful APIs. As all of the team members will have to develop or use these APIs, we find that it is appropriate to establish some brief guidelines for development. In particular, we require that these APIs are well documented such that it is clear what input is expected and what the format of the output is.

In order to fulfil this requirement, we will be using the API documentation tool Swagger[13] for all of our APIs. In addition to standardising the design of our APIs across our architecture, Swagger can generate a graphical user interface for each of our APIs, wherein it is possible to explore the different resources of an API without having to use additional third-party tools.

5.4.11 CORS

When hosting APIs it is often necessary to setup Cross-Origin Resource Sharing (CORS). CORS is a protocol that ensures that an origin is trusted to receive certain information or resources. The way CORS works differs slightly depending on the way it has been setup by the server hosting the API implementing CORS. The way it is used on our development server is very quick to setup though not very secure. In an actual production environment additional setup should be done to properly configure CORS. For our development server we used the wildcard '*' for CORS to allow any origin to access the hosted APIs. This is not considered good practise for systems that either works with sensitive information or requires authentication. Therefore it is considered good practise to setup CORS to allow only the trusted origins of the servers in the system infrastructure.

On the client side, the browser is responsible for honouring the CORS protocol. A browser adhering to the CORS protocol will, under certain circumstances, perform a preflight request, in essence asking the server "Can I perform this request?", to which the server sends a response containing whether or not the response is allowed as well as information about which HTML headers- and methods are allowed to be used by the browser.

If the server hosting the API does not give an acceptable response to the preflight request the browser will not allow further communication with the server as a security measure. The reasoning behind this decision is the fact that an incorrectly configured server can potentially be malicious to the user [11] [37].

5.4.12 Deployment

Deployment of our system will be modular to the point where each service can be deployed on different servers, or clusters of servers. Depending on the deployment, additional setup may be required. For example, in order to setup each API on different servers it is beneficial to setup some kind of web server, such as Nginx or Apache, in order to keep up with requests. Whereas if multiple APIs were setup on the same server, all of them could be served by the same web server. On the other hand, having them split between multiple servers means that in case one server goes down, the remaining servers can pull the load while the unresponsive server is being serviced or replaced, thus

[13]Swagger (`https://swagger.io/`)

eliminating downtime as long as the remaining servers can pull the load.

5.4.13 Monitoring

With a web application there are several issues that might arise which can negatively impact the user experience, and by extension the business. These issues can range from complete server failure resulting in the application being inoperative, to a surge in user requests causing delays as the server struggles to keep up. It is therefore important to be able to monitor the operational status of your web application and servers, to be able to take action as soon as an issue appears, ideally before the issue affects any users. There exists a multitude[14] of commercial solutions to monitor a variety of aspects from response time to server load and many other metrics.

With the wide variety of available monitoring solutions it is possible to tailor the monitored metrics to our application. At the user interaction it makes sense to monitor the response times, error rate and downtime. As our solution is designed to scale out it makes sense to monitor the server load. Should the server load grow too high we can then easily add additional resources to accommodate the increased load.

5.5 Data storage strategy

This section describes our overall data storage strategy. First, we will consider which entities have to be stored in our database and determine any relations between them. Next, in subsection 5.5.3 we consider various DBMSs, both relational and non-relational, and decide on which ones to use for our services, based on how these systems support the structure of our data and our requirements. Finally, in subsection 5.5.3 we consider and decide on a method for abstracting over data querying in order to ease the development process.

5.5.1 Data overview

In this section we will review the different types of data we need to store, including their structure and relations to other data types. These data types are largely inferred from the functional requirements laid out in section 4.1, but the structure of these data types will have to be considered with relation to some of the non-functional requirements.

Accounts

As our system will need to support multiple accounts, the system will need to keep track of account details and particularly account credentials. Beyond these credentials and whichever relations the account has, we will not care about storing further account details.

[14]Monitis lists a few software solutions for monitoring, http://www.monitis.com/blog/11-top-server-management-monitoring-software/ (Monitis is a cloud based monitoring solution)

Brands

One of our functional requirements states that the users of the system should be able to track more than one brand. As such, we will need to keep track of brands in the system and include some relationship between brands and user profiles.

Brand synonyms

Just as a user can be associated with many brands, a brand can be associated with multiple synonyms or search keywords. These synonyms are then used by whichever back-end crawler we implement in order to decide which SMPs should be analysed. Note that these synonyms can also be referred to as search keywords, however we have chosen not to name them as should in order to avoid any ambiguities with the KE model.

Social media posts

SMPs will need to be handled by our system, such that our SA and KE modules can extract useful data from them. An important question is whether we want to store entire messages for every SMP we store in the database. Obviously, this is not feasible at all, as we cannot expect to replicate the contents posted on the select social media. As a consequence of this choice, users of the system will not be able to view texts from mentions about their brands.

This is mostly a question about the granularity of sentiments towards individual brands and in our functional requirements, our approach to this granularity question is that the user should be provided with an overview of subtopics that users are talking about, i.e. an abstraction over individual SMPs. However, we do recognise the usefulness of being able to view the content of individual SMPs mentioning one's brand, e.g. if a brand reputation manager wishes to respond to some SMP. As such, a compromise could be to store an additional identifier (e.g. a link) for each SMP, which can then be used by a user of the system to view the source post.

In conclusion, the actual contents of SMPs will only be available within the primary storage of the system as long as it has not been handled by our models. At no point, however, will the actual contents of the SMPs be written to secondary storage.

To capture the relationship between SMPs, brands, brand synonyms and users, one can refer to Figure 5.7 for an overview. As can be inferred from the diagram, it is possible that multiple brands with the same name can co-exist in our system. However, to avoid doing redundant work, the analysis module should consider that a single search keyword may be associated with multiple brands. As can also be inferred from the diagram, SMPs only exist in our system if there is some search keyword mentioning. The drawback of this design is that a newly added brand will not have have any associated SMPs. On the other hand, we would have to analyse all SMPs on our tracked platforms, which is unfeasible due to hardware constraints and our non-functional requirements.

Analysis results

Following the functional requirements in section 4.1 and section 5.2, we should be able to present a time-series of analysis results regarding one or more brands to a user of the system. Following the

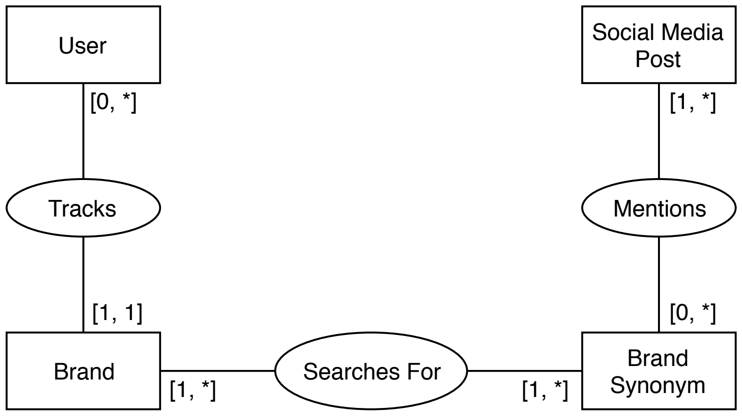


Figure 5.7: Relationship between the different entities described in this section.

requirement of a scalable system (see section 3.3 and section 4.2), we must be careful when deciding what and how much data to store.

Statistics

We assume that a user of the system is only interested in the results of the analysis, and not the individual SMPs themselves. As such, once we have conducted a scheduled analysis of a corpus for a brand, the results can be stored and the corpus discarded. This requirement means that we need some data structure to support these results. An important property of this data structure is that it will need to support a diverse range of granularities when statistics are aggregated. Furthermore, the data structure should support a possible changes to the sentiment classes, e.g. if we were to introduce new sentiment classes in the future.

In order to facilitate this aggregation of statistics, our design is an intermediate representation of statistics which, on its own, provides statistics for some synonym within an arbitrary time span. We will refer to this class of intermediate representations as *snapshots*. As snapshots should support arbitrary time spans, they should intuitively have a start time and an end time. Since the snapshot is provided for some synonym, it should also have a link to that synonym. A JSON-like version of the structure of snapshots can be seen in Code fragment 5.1. In this example, statistics are provided for two classes, i.e. negative and positive, but the design allows for an arbitrary amount of classes. To ensure backward compatibility, a query over snapshots with differences in classes would simply provide statistics for the intersection of the classes.

```
{
  "synonym": string, // synonym associated with the snapshot
  "spans_from": date,
  "spans_to": date,
  "sentiment": double, // average sentiment all posts
  "statistics":
  {
    "negative":
    {
      "num_posts": int, // number of negative posts
      "keywords": [] // top-5 keywords for negative posts
    },
    "positive":
```

```
    {
      "num_posts": int, // number of positive posts
      "keywords": [] // top-5 keywords for positive posts
    }
  }
}
```

Code fragment 5.1: Structure of the snapshot data structure.

5.5.2 Choice of DBMS

In this section we will decide on which DBMS to use for all the components of our system which require data storage. For the sake of simplicity, we will be limiting the discussion to three approaches: PostgreSQL, MongoDB and Citus. Even though the range of DBMSs spans greatly beyond these three, we feel that they differ enough to give grounds for a comparison.

Factors to consider

When considering which DBMS to use, we are more concerned with read speed than write speed. SMPs and snapshots will need to be stored regularly and could possibly be saved more frequently with write speed being prioritised, but we want to favour performance from the user's perspective in this case, even if it means that we cannot analyse posts in real time.

The DBMS will need to be able to store unstructured data. As described in section 5.5.1, snapshots contain statistics which may be extended with additional classes in the future.

Another factor to consider is that of development time. As we have limited development time with the DBMS not being the main focus, we need to choose carefully such that not too much time is spent learning the DBMS and designing the schema.

Finally, we will consider how the different DBMSs handle scalability, both in terms of space scalability and the ability to handle an increased load in reads.

PostgreSQL

PostgreSQL[15] is an open source relational DBMS which uses the SQL language. It uses a strongly typed schema, resulting in a reduced possibility of errors for developers. Relations between tables allow for schema normalisation which avoids data replication, leading to a reduced database size compared to document-store databases. Despite having highly structured schema, unstructured data is possible with the JSON data types, although updating key-value pairs means having to overwrite the entire cell. From a scalability point of view, PostgreSQL supports master-slave data replication, but otherwise its built-in features for scaling are limited.

[15]See: `https://www.postgresql.org/`.

MongoDB

MongoDB[16] is an open-source document-oriented database. Unlike the strongly typed schemas used in PostgreSQL, MongoDB uses a schemaless structure where data is stored in a binary-encoded version of JSON called BSON. From a development point of view, the schemaless design allows for rapid and iterative development without having to do much upfront design. Data is stored in documents, corresponding to tables used in relational databases. Instead of having relations between documents, related data is often stored in the same document. This means that joins are often unnecessary, but the denormalised design may lead to increased database sizes. In terms of scalability, MongoDB has built-in support for automatic sharding, which is a method for horizontally scaling out the database by distributing data across multiple machines.

Citus

Citus[17] is not a DBMS on its own, but rather an extension to PostgreSQL which relies on APIs provided by the DBMS. Whereas PostgreSQL on its own offers limited support for horizontal scaling, Citus effectively adds sharding to the DBMS. SQL queries are then processed in parallel across the servers which the data is distributed to. The architecture is similar to that of Hadoop, where one master node contains meta data about shards and uses that data to fragment incoming queries. Being an extension to PostgreSQL, migrating from single-node PostgreSQL to Citus is a relatively easy process supported by underlying migration tools in Citus.

Summary and choice of DBMS

A summary of the descriptions of the DBMSs and how they relate to our factors can be seen in Table 5.4.

	PostgreSQL	**Citus**	**MongoDB**
Read speeds	Good for highly relational data		Good with denormalised design
Space usage	Lower due to normalised schema		Higher due to denormalised schema
Unstructured data	Supported, but requires overwriting entire cell		Supported
Development time	High amount of upfront design		Design is easily developed iteratively
Developer familiarity	High	Low (high with underlying DBMS)	Low
Scalability	Medium, supports data replication	High, built-in support for sharding	

Table 5.4: Summary of DBMS comparison.

Based on our comparison, we find that Citus is the best fit for our case, especially for our gateway service since it is highly relational. Our temporary store for SMPs and permanent store for snapshots

[16]See: `https://www.mongodb.com/`.

[17]See: `https://www.citusdata.com/`.

could arguably fit just as well with MongoDB, but by choosing a single DBMS for all services we can shorten the developer time spent learning a new DBMS. Given the ease of migrating from PostgreSQL to Citus and our familiarity with PostgreSQL, we will initially be using PostgreSQL across all services and migrate to Citus if we find the time to do so.

5.5.3 Object-relational mapping

Based on past experiences, we will be using SQLAlchemy Python toolkit as our ORM in this project. SQLAlchemy will serve as an abstraction over the data in our databases and will allow us to query, insert, and manipulate rows in the database by manipulating Python objects.

In SQLAlchemy, each table is represented as a Python class. Each class defines a set of properties that, in terms of the ORM mapping, corresponds to columns in the table. In SQLAlchemy, creating new tables is a straightforward process and is not much different from creating standard classes in Python. This facility also allows us to model inheritance relationships in the database in a straightforward manner.

We choose to use an ORM in order to reduce to often cumbersome work associated with constructing SQL queries for getting the data we want from the database. We expect that, through using SQLAlchemy, the implementation phase will be less cumbersome and will increase code readability.

5.6 User interface

In subsection 5.4.4 we decided Angular would be the best fit front-end application framework for building the front-end application. This section will cover the design decisions made in the process of building the front-end application.

An angular application always starts with a root component, usually named the app component. The root component is the starting point of the application and its template serves as the basis for the application, which may include multiple sub components. An example is shown in Figure 5.8.

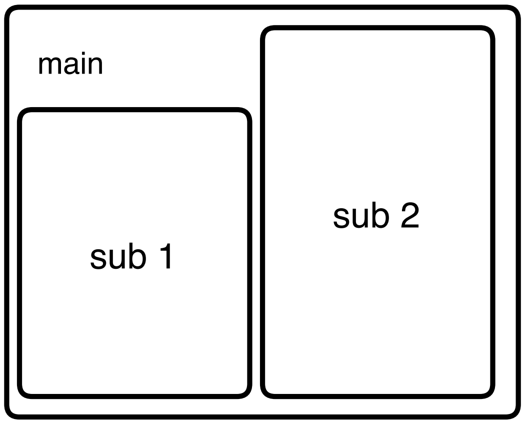


Figure 5.8: Angular hierarchy: main component with two sub components

5.6.1 Component design

Figure 5.9 shows the component design of our user interface. In this section we will go over each component, and describe their relations starting with the app component.

The app component is solely responsible for determining whether a user is authenticated or not and displaying the appropriate component based on the fact.

The unauthenticated component will display the login component as default from which it is possible for the user to navigate to the registration component should he or she so desire. The authenticated component on the other hand, will display the brands component.

The brands component will display a list of brands associated with the authenticated user, with the possibility of adding and removing brands. Each brands name should be a hyperlink to the brand component for that specific brand.

The brand component will supply data to and display the synonyms, bar chart and line chart components. The synonyms component displays a list of synonyms associated with a brand. The line chart and bar chart components will depict the statistics associated with a brand and its synonyms.

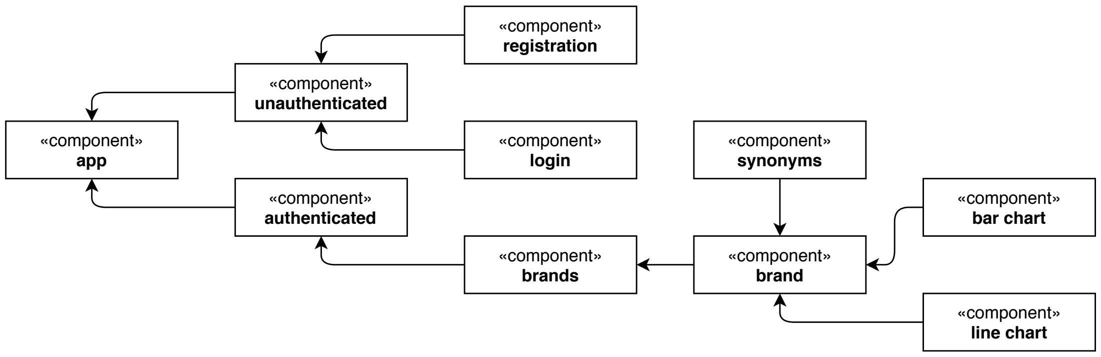


Figure 5.9: User interface component design. The arrows indicate the hierarchy of the components.

5.6.2 Service design

In order for the components to obtain data to be displayed from the gateway API described in subsection 5.3.2, we need Angular services that the components may invoke. In this section we will go over which components can invoke what services as shown in Figure 5.10.

The authorisation service is responsible for authorising a user as well as checking whether a user is authorised or not. The service is used by the app component and the login component. The app component uses it to check whether a user is logged in or not, to determine what component should be displayed to the user, whereas the login component uses it to obtain authorisation when supplied with user credentials.

The account service is responsible for creating a user account, it is used by the registration component to create a new user account.

The brand service is responsible for fetching brands, its synonyms and performing Create, Read, Update, Delete (CRUD) operations, it is used by the synonyms, brands and brand component. The synonyms component uses the service to fetch, add and delete brand synonyms; the brands component uses the service to fetch, add and delete brands; and the brand component uses the service to fetch brand data from a supplied brand id.

The statistics service is responsible for fetching statistics related to a brand. It is used by the brand component to fetch brand statistics, which the brand component then passes on to the line and bar chart components.

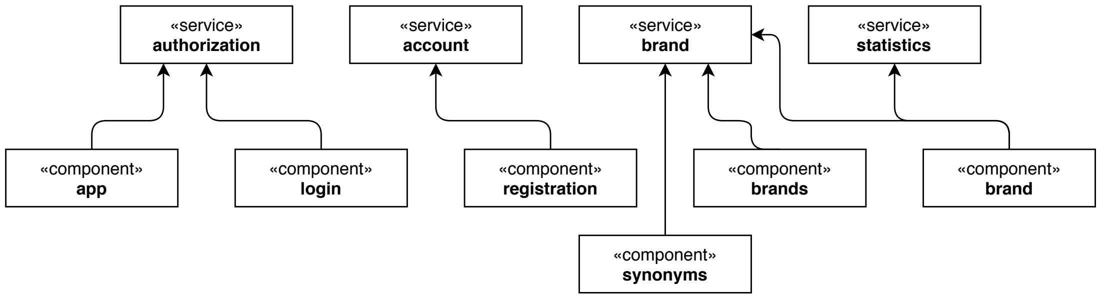


Figure 5.10: User interface services. The arrows indicate invocations.

5.7 Sentiment analysis

This section covers some of the contemporary methods for conducting SA with both lexical based and ML based methods, including a comparison of these methods in terms of their complexity and performance. Finally, we decide on a model to use in our system in subsection 5.7.3.

5.7.1 Lexical-based analysis

Lexical-based methods leverage the existence of sentiment-representative words (e.g. "angry", "happy" and "disappointed") to determine the overall sentiment of a piece of text. One example of such methods is Linguistic Inquiry and Word Count (LIWC) [48] that extends the binary classification of emotionally loaded words (i.e. positive or negative) to include 100 word categories that each represents a specific emotion or intent. Such approaches rarely succeed in capturing the underlying subjectivity and opinions towards separate entities [8, p. 13]. Furthermore, lexical-based methods are hard to design for use in differing contexts; the dictionary of sentiment-representative words is different for English literature compared to typical social media posts [20, p. 28]. With that being said, LIWC has performed well in terms of polarity classification on social media like YouTube and Twitter, while it did not perform as well as learning-based models on longer and more verbose texts from news sites such as BBC [48, p. 33].

5.7.2 Machine-learning-based analysis

Conducting SA with an ML based approach can be actualised with several different learning models. This section will briefly discuss some of these models.

All our models will utilise the word2vec, to have a consistent representation. This might mean that some models have an unfair advantage. The purpose of this report is not to achieve the best model, but one that perform fairly well, such that we given enough data are able to extract the best keywords for a given sentiment.

Naive Bayes networks

A relatively simple approach to sentiment classification is by use of a Naïve Bayes (NB) classifier as described in section A.3. Although NB is a simple approach, it yields a great accuracy. The main drawback of the method is that it assumes conditional independence between features, hence is it referred to as naive. For a document, such features could be individual words extracted from the document. In a real life setting, we cannot consider the words from a document to be independent since they can have dependencies on each other, both syntactic and semantic. For example, the word "good" on its own could be indicative of a positive sentiment, yet if the word "not" appears before "good" it could be indicative of a negative sentiment. Despite the oversimplification, NB has been shown to yield very accurate results [18][16].

SentiStrength

SentiStrength [20], [49] was created in order to capture implicit sentimental connections between features and take advantage of ML models' ability to adapt to different contexts. This model combines techniques from several other methods and successfully captures meaningful connections between text features by introducing "booster" words such as *very* and *somewhat* that affect the sentimental value of the following words. It also includes emoticons and deliberately misspelled words in its evaluation. Given a large and diverse labelled data set, ML-based models such as SentiStrength have performed very well in contexts ranging from short-text social media posts on Twitter to long and verbose texts such as news articles from BBC [20, p. 29-33]. Interestingly, models such as SentiStrength are typically based on and extends the word categories from lexical-based methods such as LIWC in order to increase classification accuracy.

Deep learning models

While SentiStrength has done well in capturing meaningful connections between features local to each other in a piece of text (see section 5.7.2), deep learning models like Convolutional Neural Networks (CNNs) (see section A.1 for description of CNNs) have been shown to capture similar connections and dependencies across sentences and reach sentiment classification accuracy comparable to that of SentiStrength [46]. For example it was possible to have an accuracy of 88.3% on IMDB reviews using a Neural Network (NN) and a two class problem using CNN and Long Short Term Memory (LSTM) (see section A.2), referred to as ConvLstm [21]. Although other deep learning models have been constructed by relying on the concept of Sentiment Propagation, which is the assumption that the intrinsic value of linguistic elements propagate throughout the syntactic structure of a sentence [1, p. 1107].

A major common drawback for all supervised learning-based models is that they require a substantial labelled data set in order to perform well [40]. Unless a large and diverse labelled data set exists, deep learning-based methods are impractical to implement [20, p. 28].

5.7.3 Choice of approach

Method	SentiStrength	LIWC	ConvLstm	Naive Bayes
Accuracy	0.815 (0.843)	0.675 (0.690)	0.883	0.82 (IMDB)
Implementation complexity	High	High	Intermediate	Easy

Table 5.5: A table showing the relations between models. Implement complexity is their apparent complexity.

We will try and implement machine learning models, as it has been shown that simple Deep Neural Networks (DNNs) can outperform most of the-state-of-the-art NLP processing solutions [53].

As shown in Table 5.7.3, ConvLstm [21] has the best accuracy and is relatively easy to implement ourselves. The accuracy is not directly comparable between ConvLstm and the two others, as the un-parenthesised number for SentiStrength and LIWC is an average accuracy for both long and short texts [20], while it is only for short texts for ConvLstm. The number in the parenthesis is the F-measure for twitter texts.

Creating the LIWC lexicon would require a lot of work to create and group terms together, as the team behind LIWC used human judges to generate their Power dictionary [32]. This would be way to expensive for us to do, it does not perform so well in terms of accuracy compared to the two other methods. The same is the case for SentiStrength, though on a smaller scale, where approximately a 1000 terms is classified as positive or negative by human judges and later modified through training phase [50].

We will use a ML based model for SA, using a deep learning model as it has been shown to be a good alternative to traditional methods and performs well [5]. [21] describes the ConvLstm, which is a rather simple model and still perform well for a two-class classification problem of short texts. We will use this as a base for our models and try to achieve similar results. Furthermore, we will implement Naive Bayes classifier, other simple classifiers as baseline models. When choosing which model to use, we will both look at their performance in terms of accuracy and throughput, where we are trying to have a high accuracy and throughput.

As described, it requires a large data set to train the deep learning model to perform well. Using Google's search engine for data sets we found a set consisting of 1.6 million short texts with a positive/negative label, meaning we have 800,000 examples per class. This data set has been used by [5] to train their model, and therefore should be sufficient.

5.7.4 Model design

This section will describe the design of the different models we will implement, how they work, and the thought process behind it.

Recurrent neural network

As described in [52], it is possible for a single Recurrent Neural Network (RNN) to achieve good results on multiple NLP tasks. We will therefore implement LSTM and Gated Recurrent Unit (GRU) without any other part as shown in Table 5.7.4, though we might introduce dropout network to reduce overfitting and help the network generalise. From this point forward we will use RNN to describe our GRU and LSTM. The RNNs is described in detail in section A.2.

The recurrent neural networks are good at finding long term dependencies in texts [52]. As a sentence might have a negative or positive word connected to some event later, to correctly predict this it has to have information from an earlier state. They are therefore a natural candidate for sentiment classification. The RNNs will have a word2vec embedding layer, looking up word vectors, such that similarity between words is the distance between the vector representation [21].

Model	
LSTM	GRU
Words	
Embedding Layer	
x hidden units	
Dense output 1	
Sigmoid activation	

Table 5.6: The RNN models we will try to implement. x is the number of hidden units, which we will use for tuning.

Convolutional neural network

CNNs extract local features as described in section A.1, which means it can extract meaning from key-phrases and determine the sentiment [52]. We will try using pooling and striding over two, as striding seem to be able to replace pooling [25]. Furthermore, simple CNN models seem to perform almost as well as state-of-the-art models [25]. We will implement the models defined in Table 5.7.4. We will try a few different x and y values (therefore also changing z) and choose the best. The models are based on [25].

Convolutional combined with Recurrent network

[21] describes a model using CNN and LSTM. We will try and run the best performing CNN of the models described in section 5.7.4, removing the three last layers in Table 5.7.4, and insert the best performing RNN model after to see if this improves our results. The reason is that CNN would extract the most important features in a keyphrase, and the RNN would then be able to learn the long term dependencies given these features. As the RNN uses the temporal information from the data, it does not make sense to minimise the output of the CNN as the last six layers in Table 5.7.4 does. Instead the RNN would be placed instead of the six layers. Furthermore, we will add a dropout layer between the CNN and RNN, as in [21], to ensure the RNN does not overfit.

Models		
Strided-CNN	ConvPool-CNN	All-CNN
Sentences with fixed length		
Embedding layer		
y field width, x feature maps	y field width, x feature maps	y field width, x feature maps
y field width, x feature maps, stride 2	y field width, x feature maps	y field width, x feature maps
	y field width, x feature maps	
	y max pooling	y field width, x feature maps, stride 2
y field width, z feature maps	y field width, z feature maps	y field width, z feature maps
y field width, z feature maps, stride 2	y field width, z feature maps	y field width, z feature maps
	y field width, z feature maps	
	y max pooling	y field width, z feature maps, stride 2
y field width, z feature maps		
1 field width, z feature maps		
1 field width, 1 feature maps		
Global average pooling		
Sigmoid		

Table 5.7: A table describing the different CNN models we will implement. y is the receptive field width that remain constant. x is our output dimension or the number of feature maps, where z is the same, but $z = x * 2$.

Naive bayes

As the NB achieve results comparable to that of some ML models, we want to use it as a baseline. The NB classifer (described in section A.3), where we use the multinomial model[18] [47]. A vocabulary should be created from the train data, such that the input to the NB classifier can be defined as a vector with the number of occurrences of that word. For example given a vocabulary

$$V = \{the, dance, naive, bayes\}$$

then the sentence '*the naive model is the best*' would generate a vector of the size of the vocabulary, such that the first element in the output represents the number of occurrences in the first word in the vocabulary in the text, as

$$(2,0,1,0)^T.$$

This model therefore ignores order of words in the document and therefore any dependencies between the words. It is therefore also a great model to check if our NN models learn any dependencies.

5.8 Keyword extraction

Part of the solution to the problem stated in section 3.4 requires that we can successfully implement KE; we must be able to extract keywords- and phrases from SMPs so they can used to provide context for the calculated sentiment. Following the brief description of different approaches to

[18]Formula described here `https://www.inf.ed.ac.uk/teaching/courses/inf2b/learnnotes/inf2b-learn07-notes-nup.pdf`.

keyword extraction in section 3.2, this section covers these approaches in greater depth. We briefly cover the design of the KE approach when using Term Frequency-Inverse Document Frequency (TFIDF) as described in subsection A.4.1 and cover the graph-based approach when using KGs as described in subsection A.4.2 in depth in subsection 5.8.2. For the KG approaches, we consider a set of different configurations based on our own extensions to the base approach described by [2] for KE using KG. Our modifications to the base model, including the reasoning behind them, are described in depth in subsection 5.8.3.

As previously mentioned in chapter 3 and section 3.4, our system must be both functionally sound and scalable. As such, it is likely that we need to incur a trade-off between performance w.r.t. keyword quality and extraction time. For this purpose, the performance of the approaches, including their running time and precision, is evaluated and compared in section 7.4.

First, we will define what a keyword is in the context of SMPs. Our definition is inspired by the general definition of keywords over text given by [2].

> A **keyword** is the smallest textual unit or sequence of units that best represents a common feature over a set of SMPs.

Note on machine-learning KE approaches

Two different approaches for KE, namely feature-based extraction and graph-based extraction, are covered in detail in section A.4. An entirely different set of approaches fall under that of supervised ML approaches, such as using NB [51]. However, we have chosen not to include ML approaches in this project. The reason for this is because we would prefer our keyword extraction module to be agnostic towards the contents of the SMPs gathered by the system. A supervised machine learning model requires labelled training data and will, during training, seek to optimise according to some objective function. Naturally, while ML models can potentially infer both temporal and structural dependencies between words in text, the trained model will be fit to the training data - specifically the words and writing style of that used in the training data. We expect that both the writing style and words used in the text we gather will vary, and more importantly will change over time. Instead, unsupervised approaches like the ones described in section A.4, while not necessarily being able to capture high level features in the way that ML models can, will always be able to extract keywords from a corpus regardless of the words used and the writing style, as the extraction of keywords is entirely based on the "structural" correlation between words in a text.

5.8.1 Design of keyword extraction using TF-IDF

Given a collection, D, of SMPs regarding the same brand or synonym. We refer to each of these as separate documents $d_1, d_2, \ldots \in D$. First, the data is preprocessed. The preprocessing algorithm is the same for TFIDF and KG and is described in Algorithm 1.

When extracting keywords for a brand over a set of synonyms, we first calculate a TFIDF score for every word w in every document $d \in D$:

$$Score(w,d) = \{(w, TFIDF(w,d,C))\}$$

Algorithm 1 Preprocessing

```
1: function PREPROCESSTEXT(text, negate, stopwords, lemmatize)
2:     $text \leftarrow$ CLEANTEXT($text$)                    ▷ Removes non-textual information.
3:     $words \leftarrow$ TOKENIZE($text$)                    ▷ Tokenize every word.
4:     $words \leftarrow$ NORMALIZE($words$)    ▷ Lowercase words and remove non-ASCII characters.
5:     $words \leftarrow$ REMOVEPUNCTUATION($words$)
6:     if $stopwords == False$ then
7:         $words \leftarrow$ REMOVESTOPWORDS($words$)
8:     if $lemmatize == True$ then
9:         $words \leftarrow$ LEMMATIZE($words$)
10:    return $words$
```

... where $TFIDF$ is defined as in subsection A.4.1.

After calculating the TFIDF scores for all words, the top-k scoring words are extracted as keywords for the document d.

This process is repeated for every document $d \in D$. When all documents have been processed, we choose the k keywords that were chosen the most frequently from the set of documents D as the keywords for the entire corpus. The approach is described algorithmically in pseudocode in Algorithm 2. We keep track of the TFIDF scores in a 2-dimensional matrix M where we can access the TFIDF of a word w in a document d with the following notation: $M_{d,w}$. Once all words in all documents have had their TFIDF calculated, top-k keywords are chosen from each document. Then, the keywords for each document are compared, and the k keywords that reappear most frequently across all documents are chosen as the keywords of the corpus.

Algorithm 2 Pseudocode for extracting k keywords from a corpus D using TFIDF.

```
1: function TFIDFKEYWORDS($D, k$)
2:     $W \leftarrow \{w|w \in d, d \in D\}$                    ▷ Set of all unique words
3:     $M \leftarrow [D][W]$                          ▷ 2-dimensional matrix, 0 in every cell
4:     $KWs \leftarrow []$                            ▷ Dictionary from words to frequency
5:     for $d \in D$ do
6:         for $w \in d$ do
7:             $M_{d,w} \leftarrow$ TFIDF($w, d, D$)
8:     for $d \in D$ do
9:         $scores \leftarrow$ ZIP($W, M_d$)              ▷ Get TFIDF scores for all words in $d$
10:        SORT($scores$)
11:        $Top_k \leftarrow \{(w_n, s_n)|(w,s) \in scores,\ 0 \leq n \leq k\}$        ▷ Take the top-$k$
12:        $KWs$.UPDATE($Top_k$)     ▷ Increment the frequency of the chosen keywords
13:    SORT($KWs$)                     ▷ Sort the keywords chosen for all documents
14:    $Top_k \leftarrow \{w_n|w_n \in KWs,\ 0 \leq n \leq k\}$            ▷ Take the top-$k$ keywords
15:    return $Top_k$
```

5.8.2 Design of graph-based keyword extraction

The second graph based approach is heavily based on the Twitter Keyword Graph (TKG) introduced in [2]. The approach is described formally in the following. Given a set of documents $d_1, d_2, \ldots d_i \in D$ each representing a single SMP, we consider the corpus D as a collection of documents regarding the same brand. First, the corpus is preprocessed using the `PreprocessText` function seen in Algorithm 1.

Following the preprocessing, we end up with a set of token streams $T_{stream} = \{T_1, T_2, \ldots T_i | T_i = \text{ a stream of tokenised words from } D_i\}$ and a token set $T_{set} = \{t | t \in D_1 \cup D_2 \ldots \cup D_i\}$. For every token $t \in T_{set}$, a graph node is created. When all nodes have been created, edges are created between nodes if they co-occur in the same k-length sequence of tokens in the same document. This is referred to as k-neighbour edging as described in [2]. The weight of all edges in the graph is set to 1 regardless of frequency of co-occurrence. The process of building this graph can be seen in Algorithm 3.

Once the graph $G = (N, E)$ has been built, the top-k keywords are extracted by calculating the closeness centrality $Closeness(n, G)$ of every node $n \in N$ in the graph (see subsection A.4.2 for definition).

Algorithm 3 Pseudocode for building a KG from a corpus "tokenStreams" using k-neighbour edging.

```
1:  function BuildGraph($T_{stream}$, $T_{set}$, $k$)
2:      $G \leftarrow (N, V) | N = T_{set}, V = \emptyset$
3:      for $stream \in T_{stream}$ do                          ▷ Split stream into shingles of $k$ length
4:          $shingles \leftarrow \{(t_i, \ldots t_{i+k} | t \in stream, 0 \leq i \leq |stream| - (k-1))\}$
5:          for $s \in shingles$ do
6:              $first \leftarrow t_0 \in s$
7:              for $neighbour \in s \setminus \{first\}$ do
8:                  if $(first, neighbour) \notin E$ then
9:                      $E \leftarrow E \cup \{(first, neighbour)\}$
10:     return $G$
```

The result of the centrality calculation is a list of node/closeness pairs $P_C = \{(n, Closeness(n, G)) | n \in T_{set}\}$. The list is sorted on the closeness centrality measure and the top-k elements are returned. This process can be seen in Algorithm 4.

Algorithm 4 Pseudocode for extracting k keywords from a KG G.

```
1:  function ExtractKeywords($G = (N, E)$, $k$)
2:      $P_C \leftarrow \{(n, Closeness(n, G)) | n \in N\}$
3:      $P_C \leftarrow Sort(P_C)$                              ▷ Sort based on centrality score
4:      return $\{n_1, \ldots n_k | (n_i, c) \in P_C\}$
```

We have further extended the TKG approach described here with a set of different configurations. These modifications of the TKG approach are described in subsection 5.8.3.

5.8.3 Modifications of the graph-based approach

In the following, the graph-based approach to KE will be referred to as KG. The base approach is altered both in terms of edge weighting strategies and centrality measures, and we will distinguish between the approaches with the following naming scheme.

Every graph-based approach will be named according to the following scheme:

```
KG | CENTRALITY MEASURE | WEIGHTING STRATEGY
```

Two different centrality measures are used: closeness centrality (denoted by C) and PageRank (denoted by P). Both of these approaches are covered in detail in subsection A.4.2. Three different weighting strategies are used:

Identical weighting, denoted by 0, assigns the same weight to all edges in the graph.

Frequency difference weighting, denoted by F, assigns, for every pair of nodes u, v between which there is an edge, the absolute difference between the frequency of the words represented by u and v, respectively. Formally, given an edge $e = (u, v)$ where u and v both represent unique words in the corpus D, the weight assigned to e, w, is defined by the absolute difference between word frequencies $F(u,v) = |freq(u,D) - freq(v,D)|$. Before assignment, the weight is normalised against the sum of weights of all edges connected to u. Formally, the weight assigned to (u, v) is the normalised absolute frequency difference $Norm(F(u,v)) = \dfrac{F(u,v)}{\sum F(u,w)|(u,w) \in E}$ where E is the set of edges in G.

Frequency difference weighting (inverse), denoted by F_I, assigns weight similar to that of frequency difference weighting, but assigns the reciprocal of the absolute frequency difference between the words represented by the nodes. Formally, given an edge $e = (u, v)$ where u and v both represent unique words in the corpus D, the weight w assigned to e, is defined by the reciprocal absolute difference between word frequencies $F_I = \dfrac{1}{|freq(u,D) - freq(v,D)|}$. Like in absolute frequency difference weighting, we normalise w against the sum of weights before assignment. Formally, the weight w assigned to (u, v) is the normalised reciprocal absolute frequency difference $Norm(F_I(u,v)) = \dfrac{F_I(u,v)}{\sum F_I(u,x)|(u,x) \in E}$ where E is the set of edges in G.

Note on the chosen weighting strategies and centrality measures

We have chosen to include more and different weighting strategies and centrality measures than those proposed in [2] on which this KE approach is based. The choices come from a mix between our understanding of what a keyword is and algorithm-specific details. Words are represented as nodes connected by edges in a KG. Our idea is that if two words are correlated by an edge, then they may share some feature from the corpus they came from. We selected frequency as the shared feature, and as such assign the absolute difference of frequencies between words as the weight of the edge connecting them in the $KG|C/P|F$ approaches. Inversely, we assign the reciprocal of this difference as the edge weight in the $KG|C/P|F_I$ approaches. For both frequency-based weighting strategies, we normalise the weight according to the sum of weights connected to a given node. While this will not affect the calculation of closeness centrality, it allows the PageRank algorithm to run correctly as the weight from one node to another will represent the transition probability.

Continuing with PageRank, we have added the algorithm (see subsection A.4.2) as a means of calculating the centrality measure in addition to that of closeness which was used in [2]. Seeing that PageRank is designed to rank web-pages according to their importance, it stands to reason that the same algorithm could perhaps be used to rank words according to their importance, too. Naturally, this assumption requires testing which we will discuss in subsection 7.4.2.

As an example, the KG approach using closeness centrality and identical weighting is referred to as $KG|C|0$

Chapter 6: Implementation

This chapter describes how we have implemented various parts of our system. Initially we describe how the server is set up in section 6.1, which includes the overall structure of the system. Afterwards, in section 6.2 we describe the UI and show some examples of the implementation, including descriptions of the components we have implemented. In section 6.3 we describe how we have implemented the SA model, and in section 6.4 we describe how one could extend the model with negated words. Finally, we describe how we extract keywords given a corpus in section 6.5.

6.1 Server

The server side implementations, the developed APIs as well as the scheduler, are made in Python using the Flask framework with Flask RESTplus (see subsection 5.4.6). As can be seen in Figure 5.1, the server components are built as self-contained services. Each service can be seen as an isolated and self-contained server that exposes a number of API endpoints. This section goes into detail w.r.t. the structure and implementation of these services.

6.1.1 API structure

Each API is structured as can be seen in Code fragment 6.1.

Code fragment 6.1: The structure of each API in the system

```
main - - controller - - (api endpoint handlers)
     |
     |
      - - model - - (models)
     |
     |
      - - service - - (api logic)
```

Model

The `model` directory contains Python files determining the structure of the data that is manipulated by a specific service endpoint. For example, take the `Gateway` service that exposes, among others, the `/synonym` endpoint that is used for creating new synonyms in the system. Within `main/model`, the file `synonym_model.py` exists and specifies the `Synonym` entity in the database. Entities are defined using the SQLAlchemy framework as the ORM to our SQL database.

Code fragment 6.2: Definition of the `Synonym` database entity in `synonym_model.py` in the `Gateway` service.

```python
class Synonym(db.Model):
    """ SQLAlchemy model for brand synonyms. """

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    # To avoid redundancy, synonyms are unique
    synonym = db.Column(db.String(255), unique=True, nullable=False)
```

Service

The `service` directory contains a set of Python files that implement the logic of a specific API endpoint. For example, when creating a new synonym, the `synonym_service.py` file implements the functionality of creating a new `Synoynm` database entry with the correct information if it does not already exist, and commits the synonym entry to stable storage. The implementation of this functionality can be seen in Code fragment 6.3.

Code fragment 6.3: Definition of the `add_synonym` function logic in the `synonym_service.py` file in the `Gateway` service.

```python
def add_synonym(brand_id, synonym_data):
    """ Add a synonym to be associate with a brand. """
    synonym = preprocess_synonym(synonym_data['synonym'])

    # Check if the synonym already exists
    # If it does not exist, create it
    existing = Synonym.query.filter(Synonym.synonym.ilike(synonym)).first()
    if not existing:
        synonym = Synonym(synonym=synonym)

        db.session.add(synonym)
        db.session.flush()

        # Refresh the current session in order to get the new synonym
        db.session.refresh(synonym)
        existing = synonym

    # Check if the synonym is already associated with the brand
    if BrandSynonym.query.filter((BrandSynonym.brand_id == brand_id) & (BrandSynonym.synonym_id ==
        existing.id)).\
          scalar():
        return dict(success=False,
                    message="The synonym is already associated with the brand."),
                        SynonymServiceResponse.AlreadyExists

    # Create an association between the synonym and the brand
    association = BrandSynonym(brand_id=brand_id, synonym_id=existing.id,
        created_at=datetime.datetime.utcnow())

    db.session.add(association)
    db.session.commit()
```

Controllers

The `controllers` directory contains a set of Python files that implement the API endpoint itself. The controllers verify the input data and calls the appropriate logic in the `_service.py` files in order to complete the request. As such, the actual API endpoints are implemented by the `_controller.py` files. The `Gateway` service implements a `synonym` API endpoint that can be used to get a listing of all active synonyms with a `GET` request as seen in Code fragment 6.4. This is a resource used by the API for handling the creation and deletion of brands.

Code fragment 6.4: Implementation of the `synonym` API endpoint in the `Gateway` service.

```
...
@api.route('')
class SynonymsResource(Resource):
    @api.doc('Retrieve a list of all the currently tracked synonyms.', security='key')
    @key_required(api)
    def get(self):
        return dict(synonym_service.get_active_synonyms())
```

6.1.2 The scheduler service

One of the most important services in our system is the scheduler service. It is through this service that crawlers are started, synonyms are added to the crawlers' search sets, and SA and KE are running as scheduled jobs.

Crawlers

This section will briefly cover the implementation of the Trustpilot crawler as designed in Figure 5.4. We will not describe the Reddit crawler as we use Reddit's own API and it is therefore not interesting to discuss. The crawler is implemented as a class object that instantiates the crawler process in a separate thread. The method responsible for instantiating the crawler thread can be seen in Code fragment 6.5.

Code fragment 6.5: The Trustpilot crawler method for instantiating a crawler thread.

```
1  def begin_crawl(self, synonyms=None, verbose=False):
2          if synonyms is not None:
3              self.add_synonyms(synonyms)
4
5          crawler_thread = Thread(target=self._threaded_crawl, args=[self.synonym_queue, verbose],
               daemon=True)
6          crawler_thread.start()
```

As seen in Code fragment 6.5, the method can be called with a list of synonyms as input. On line 3, all synonyms in the list are added to the synonym queue. Followingly, the crawler thread is instantiated as a `Thread` object with a direct reference to the synonym queue in the main thread (referenced as `self.synonym_queue`). By sharing this queue object between the crawler thread and main thread allows us to add synonyms to the crawler while it is running when, for example, a user creates a new brand with unseen synonyms. This queue object is directly implementing the frontier of the crawler, as seen in Figure 5.4.

The method implementing the logic within the crawler thread can be seen in Code fragment 6.6. As can be seen here, the crawler fetches the next `url_queue` object from the queue residing in the main thread. This is consistent with the design in Figure 5.4; the frontier is a queue where each element (referred to here as the `url_queue`) is a queue identified by a synonym that contains enqueued links that should be crawled for this synonym.

If the `url_queue` is empty, there are no more links to search through for this synonym. As such, the crawler performs a Trustpilot search query for the synonym in order to get a list of review page links for this synonym once again, in hope that new reviews have appeared in the meantime.

In `self.get_reviews_from_urls()`, the crawler parses the HTML page pointed to by the URL and extracts all text posts from the page. It also fetches the link pointed to by the `Next Page` button if it exists on the page, which is enqueued in the URL queue.

The reviews are cleaned and committed to stable storage in lines 28 and 29.

Finally, the `url_queue` is reinserted into the frontier residing on the main thread as the crawler continues with the next entry in the frontier.

Code fragment 6.6: The method implementing the crawler thread in the Trustpilot crawler.

```
1  def _threaded_crawl(self, queue, verbose=False):
2      while True:
3          try:
4  
5              # Get the next synonym dict in the queue
6              url_queue = queue.get()
7              if verbose:
8                  print(f"TrustPilotCrawler._threaded_crawl: {url_queue.tag()} retrieved from
                       synonym_queue")
9  
10             # Get the synonym of the URL queue
11             synonym = url_queue.tag()
12 
13             if url_queue.empty():
14                 # The queue should be restarted from the initial Trustpilot search.
15                 self.synonym_queue.put(self._geturlqueue(synonym))
16                 # Skip the rest of this loop
17                 continue
18 
19             url = url_queue.get()
20             # Get reviews from this URL
21             if verbose:
22                 print(f’Processing: {url}’)
23                 print(f’Currently looking at synonyms: {self.synonyms}’)
24                 print(’-------------------------------------------------------------------------------------’)
25 
26             reviews, next_page = self._get_reviews_from_url(review_page_url=url)
27             # Store the extracted reviews
28             for review in reviews:
29                 self._process_entry(synonym, review)
30 
31             # Requeue the synonym dict
32             if next_page is not None:
33                 url_queue.put(next_page)
34 
35             queue.put(url_queue)
36 
37         except Exception as e:
38             print(f’Exception encountered in crawling thread: {e}’)
39             traceback.print_exc()
40             return
```

Main schedule

The main purpose of the scheduler service is to coordinate the operations of the crawlers, SA, and KE modules in the system. The scheduler is implemented as a class with a variety of different methods for handling this process. The first method is the `begin_schedule` method seen in Code fragment 6.7.

Code fragment 6.7: The `begin_schedule` method within the Scheduler class. This method is called when the main schedule is started.

```python

def begin_schedule(self):
    ...
        # Begin crawlers
        self.reddit.begin_crawl()
        self.trustpilot.begin_crawl()

    ...
        # Begin main schedule
        self.schedule_thread = Thread(target=self._threaded_schedule)
        self.schedule_thread.start()
```

The main schedule thread is started with the `self._threaded_schedule` method as its target method. This method can be seen in Code fragment 6.8.

Code fragment 6.8: The target method of the main schedule thread. This is the method that performs the actual scheduling of services in the Scheduler class.

```

def _threaded_schedule(self):
    # Run forever
    while True:

        # Retrieve active synonyms from gateway
        self.update_synonyms(self.fetch_all_synonyms().keys())

        # Get and commit new posts from the crawlers
        self.commit_reviews(self.retrieve_posts())

        # Get and update sentiments for new posts
        posts = self.fetch_new_posts()
        print(f'{len(posts)} posts fetched')
        if posts:
            sentiments = self.calculate_sentiments(posts)
            try:
                self.local_db.update_sentiments(sentiments)
            except Exception as e:
                print(f'Scheduler._threaded_schedule: Exception encountered with
                     local_db.update_sentiments: {e}')
                traceback.print_exc()

        # If enough time has passed since the last keyword extraction, perform
        # keyword extraction and save snapshots from current interval
        if datetime.utcnow() > self.kwe_latest + (2 * self.kwe_interval):
            print(f'Current snapshot date: {self.kwe_latest}')

            for synonym in self.all_synonyms:
                snapshot = self.create_snapshot(synonym, self.kwe_latest,
                     self.kwe_latest+self.kwe_interval)

                if snapshot:
                    snapshot.save_remotely()

            # Reset KWE timer
            self.kwe_latest += self.kwe_interval
        else:
            # Wait for 10 seconds before continuing.
            sleep(10)
```

In every iteration of the scheduling thread seen in Code fragment 6.8, we first update the synonyms within the scheduler in case that a user has created a new synonym while the scheduler was performing SA or KE during its last pass. Followingly, all newly fetched posts from the crawlers are fetched from the crawlers, committed to stable storage, and removed from the crawlers' local storage. After that, all posts that have not had their sentiment calculated yet are fetched from stable storage and updated with their calculated sentiment. The method responsible for calculating the sentiment of posts can be seen in Code fragment 6.9. After calculating the sentiment of posts, we check whether or not it is time to conduct KE. At the time of writing, we conduct KE every hour, but this interval can be set arbitrarily. If it is time to extract keywords, all posts are split up in a synonym-by-sentiment fashion, after which keywords are extracted for all posts grouped by sentiment. The results of KE and SA are stored as a snapshot, the structure of which can be seen in section 5.5.1. The method responsible for creating snapshots and conducting KE can be seen in Code fragment 6.10. This process continues indefinitely, gradually filling the database with more statistics regarding all brands and synonyms as the crawlers gather more and more data.

Code fragment 6.9: The `calculate_sentiment` method responsible for updating the posts committed to stable storage with their calculated sentiment. The model used for calculating sentiment is called through the SA API.

```
1
2   def calculate_sentiments(self, posts):
3       ...
4           # Extract the post contents
5           id_list = []
6           content_list = []
7           for id, content in posts.items():
8               id_list.append(id)
9               content_list.append(content)
10
11          # Call the SentimentAnalysis API
12          try:
13              predictions = json.loads(requests.post(self.sa_api, json=dict(data=content_list)).text)
14          except Exception as e:
15              print(f'Scheduler.calculate_sentiments: Exception encountered with SA API: {e}')
16              traceback.print_exc()
17              return []
18
19          # Combine predictions with posts
20          results = [{'id': id_list[i], 'sentiment': predictions['predictions'][i]}
21                      for i in range(0, len(predictions['predictions']))]
22
23          return results
```

As seen in Code fragment 6.9, the actual SA is being done through calling the SA API on line 13. By separating the SA module from the scheduling service itself, we ensure that if the SA module crashes, the scheduler will continue to run, allowing the crawlers to keep gathering information and the KE module to keep extracting keywords for the posts that have had their sentiment calculated.

Code fragment 6.10: The `create_snapshot` method responsible for extracting keywords for posts over a given time range and returning the final result as a snapshot - the structure of which can be seen in section 5.5.1.

```
1
2  def create_snapshot(self, synonym, from_time=datetime.min, to_time=datetime.now()):
3    ...
4      statistics = dict()
5      try:
6          posts = self.local_db.get_kwe_posts(synonym, from_time, to_time)
7      except Exception as e:
8          print(f'Scheduler.create_snapshot: Exception encountered while retrieving posts from
               database: {e}')
9          traceback.print_exc()
10
11     if posts:
12         # Calculate the average sentiment over the posts in the time range
13         avg_sentiment = mean([p["sentiment"] for p in posts])
14         # Split the posts in two groups according to their sentiment
15         splits = [{"sentiment_category": sc["category"],
16                   "posts": [p["content"] for p in posts if sc["upper_limit"] >= p["sentiment"] >=
                         sc["lower_limit"]]}
17                  for sc in self.sentiment_categories]
18
19         # For each split of posts, compute keywords and number of posts
20         for split in splits:
21             keywords = []
22             num_posts = len(split["posts"])
23
24             # Only requests keywords if there are posts
25             if num_posts:
26                 try:
27                     response = requests.post(self.kwe_api, json=dict(posts=split["posts"]),
28                                              headers=self.kwe_api_key).json()
29                     keywords = response.get('keywords', [])
30                 except Exception as e:
31                     print(f'Scheduler.create_snapshot: Exception encountered with KWE API: {e}')
32                     traceback.print_exc()
33
34             # Prepare the statistics array for insertion in the snapshot object
35             statistics[split['sentiment_category']] = {"keywords": keywords, "posts": num_posts}
36     else:
37         return None
38
39     # Return the snapshot of this time range
40     return Snapshot(spans_from=from_time, spans_to=to_time, sentiment=avg_sentiment,
           synonym=synonym,
41                     statistics=statistics)
```

As seen in Code fragment 6.10, like in `calculate_sentiment`, the KE module has also been separated from the main scheduling thread and is used through calling the KE API on line 27. Once again, this ensures that all sub-processes of the scheduler can continue running were the KE module to fail.

First, the `create_snapshots` fetches all posts ready for KE from the database. Note that posts that have not had their sentiment calculated yet are not eligible for KE, as we later group the keywords by their assigned sentiment.

After grouping the posts by their sentiment into two groups, the posts are put together as a corpus of text and sent to the KE module through the KE API. The response object returned by the API call is an array of top-5 keywords over the given corpus. Finally, the results (synonym, time range, sentiments and keywords) are combined in a snapshot object (see section 5.5.1 for the structure of the object). This object is then stored in the main database for system users to query.

6.2 User interface

This section will go over the various components from the user interface component design described in subsection 5.6.1 as well as their implementation. As the code is trivial, it will not be covered, instead we will look at images of the components and go over the thought process of it all.

6.2.1 Log in and registration

The log in and registration components are the only components available to a non-logged in visitor. As can be seen in Figure 6.1 these components have a similar appearance; a card with a form and some hyperlinks. If a visitor is not logged in, he or she is greeted by the log in component. If the visitor does not have a user account, he or she may choose to click the "Register?" hyperlink, after which the log in component is replaced by the registration component. The visitor may revert to the log in component by clicking the "Already have a user?" hyperlink.

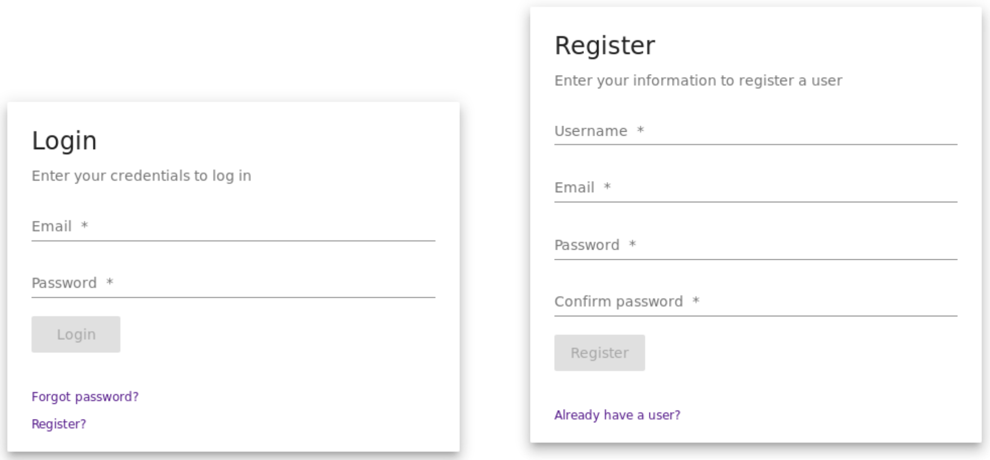
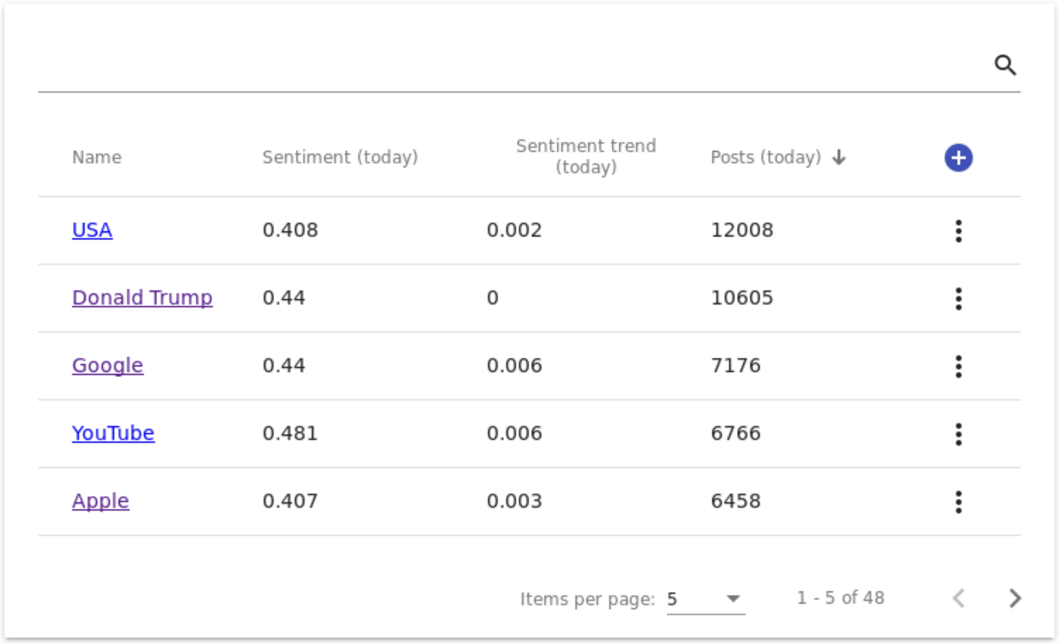


Figure 6.1: Log in and registration component.

6.2.2 Brands

The brands component seen in Figure 6.2 is a card with a table of brands, an input field for filtering the table and a table pagination bar. This component lists all of the brands a user has created. Each row has columns that can be sorted, such that the user can quickly gain insight about the brands he is currently monitoring. The name of a brand is a hyperlink to a page with more statistics about the brand in question.



Name	Sentiment (today)	Sentiment trend (today)	Posts (today) ↓	+
USA	0.408	0.002	12008	⋮
Donald Trump	0.44	0	10605	⋮
Google	0.44	0.006	7176	⋮
YouTube	0.481	0.006	6766	⋮
Apple	0.407	0.003	6458	⋮

Figure 6.2: Brands component.

6.2.3 Synonyms

The synonyms component seen in Figure 6.3 is a card with a table of synonyms for a brand, an input field for adding new synonyms and a table pagination bar. Each row has the name of the synonym and a deletion button on the far right, allowing the user to quickly add or delete synonyms for a brand.

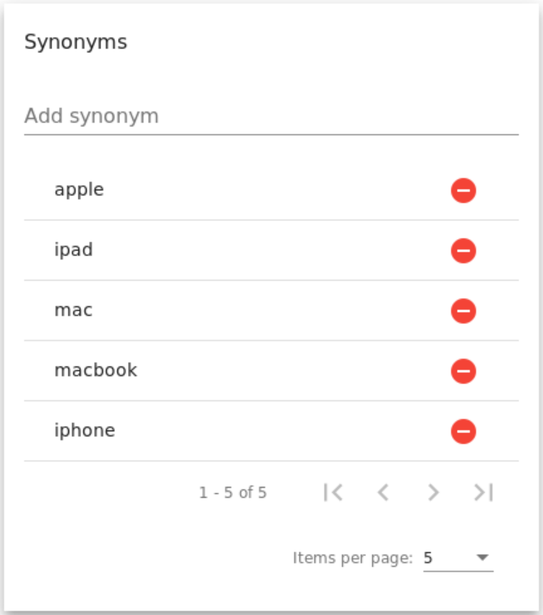
6.2.4 Charts

Currently, there are two chart components that can be used to visualise the statistics received from the statistics API (the statistics service is described in subsection 5.3.5), which both follow the same basic structure as they are both built using the ngx-charts module[1]. The line chart component seen in Figure 6.4 is used to visualise the sentiment of a brands synonyms over time and the bar chart component seen in Figure 6.5 is used to visualise the distribution of positive and negative sentiments over time. Both components must be supplied with data from the statistics API, which in turn they will convert to a data format that adheres to the data format of the specific ngx-chart.

6.2.5 Brand

The brand component is a collection of components. It includes the synonyms component from subsection 6.2.3 and the line chart component as well as the bar chart component from subsection 6.2.4.

[1]https://swimlane.gitbook.io/ngx-charts/


Synonyms

Add synonym

apple

ipad

mac

macbook

iphone

1 - 5 of 5

Items per page: 5

Figure 6.3: Synonyms component.

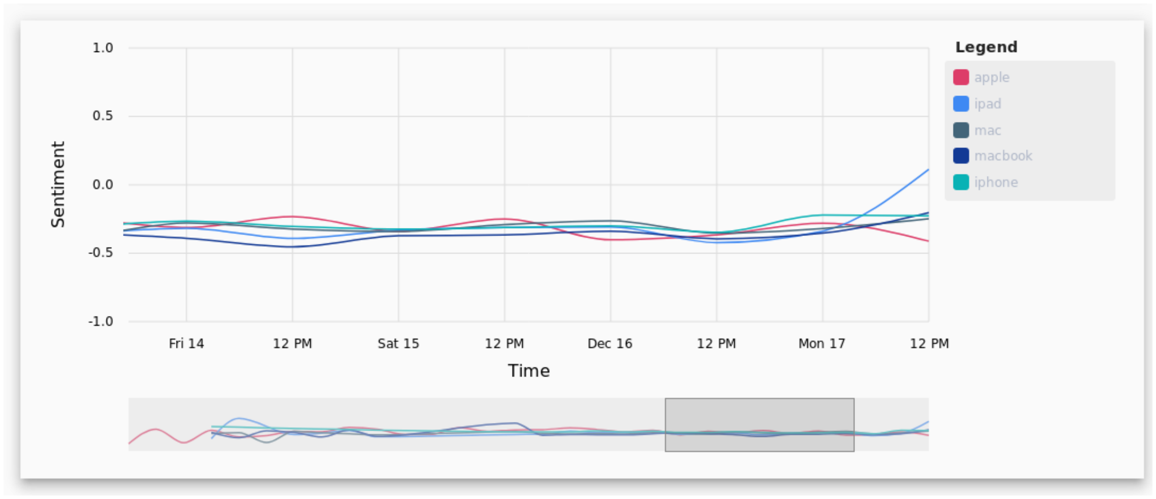


Figure 6.4: Line chart component.

The brand component thus acts as kind of a middle man, as it will pass the brand id to the synonyms component and statistics data to the charts, such that only one API request is needed in place of one for each component. It also includes a small filtering component as seen in Figure 6.6, that can be used to alter the request parameters for the statistics API.

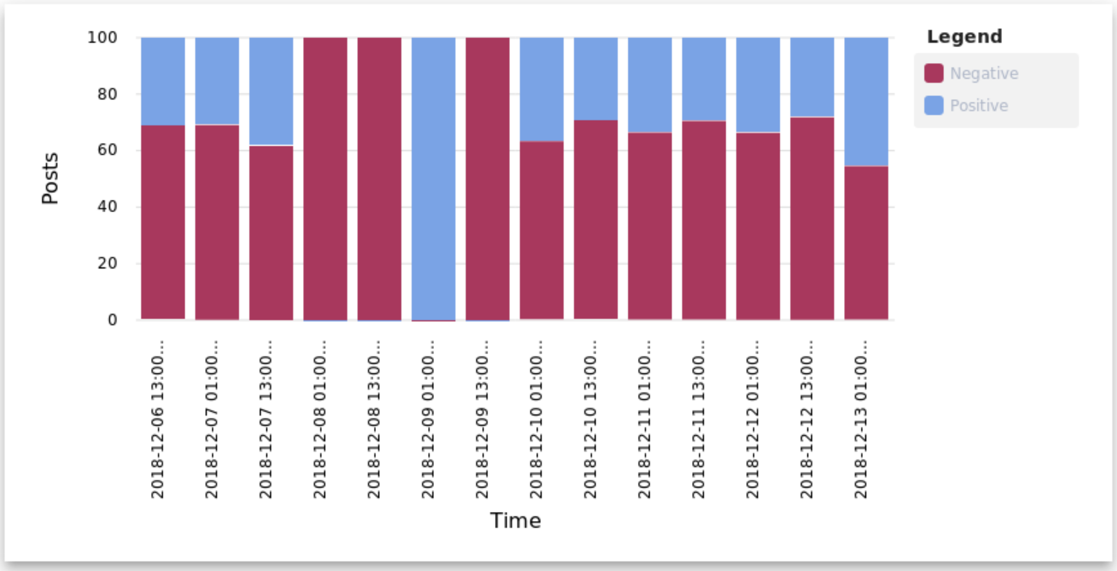
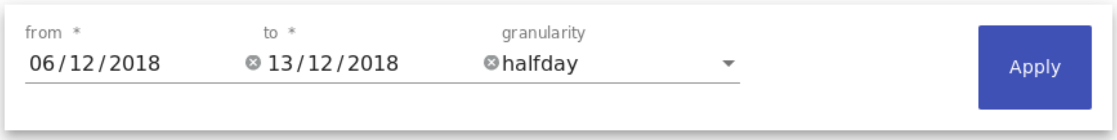


Figure 6.5: Bar chart component.



from * 06/12/2018
to * 13/12/2018
granularity halfday

Figure 6.6: Statistics filtering component.

6.3 Sentiment analysis

This section covers the implementation of the SA model described in section 5.7. To implement our SA model we have, as described in subsection 5.4.8, used Keras.

6.3.1 Vectorization

Before we can find the sentiment of a text document, we need to transform it into a vector. To do so, we used the keras Tokenizer[2] for our keras models and scikit-learn CountVectorizer[3] for our NB classifier. Although implemented differently, they both do essentially the same thing, so for simplicity we will refer to them simply as vectoriser. First, the vectoriser is fitted to the text documents in the dataset, which essentially means constructing a vocabulary of unique words across all of the text documents:

[2]https://keras.io/preprocessing/text/#tokenizer
[3]https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

$$["\text{product is good}", "\text{product is bad}"] \rightarrow \begin{array}{ccc} \$ & \mapsto & 0 \\ \text{product} & \mapsto & 1 \\ \text{is} & \mapsto & 2 \\ \text{good} & \mapsto & 3 \\ \text{bad} & \mapsto & 4 \end{array}$$

When we subsequently call the vectoriser on a text document, each word is converted to an index that corresponds to the index in the vocabulary, defaulting to index 0 if the word isn't in the vocabulary:

$$"\text{good and bad}" \mapsto \begin{bmatrix} 3 \\ 0 \\ 4 \end{bmatrix}$$

6.3.2 Embedding matrix

To use a pretrained word2vec model as a layer in our keras models, we need to construct an embedding layer that binds each unique word in the word index (see subsection 6.3.1) to the word vector in the word2vec model. This process is implemented as seen in Code fragment 6.11. First step is to get the length of the word index and instantiate the matrix with 0s, such that there are null embeddings for the words that does not exist in the pretrained word2vec model. Next step is to loop over each unique word in the word index. If the word is in the word2vec vocabulary, add the vector to the matrix at the index of the embedding matrix that matches the index of the word index.

Code fragment 6.11: Constructing the embedding matrix

```
nb_words = len(word_index)+1

embedding_matrix = np.zeros((nb_words, vec_len))
for word, i in word_index.items():
    if word in word2vec.vocab:
        embedding_matrix[i] = word2vec.word_vec(word)
```

Code fragment 6.12: Constructing the embedding layer

```
embedding_layer = Embedding(embedding_matrix.shape[0],
    embedding_matrix.shape[1],
    weights=[embedding_matrix],
    trainable=False)
```

The matrix can then be used to instantiate an embedding layer, as seen in Code fragment 6.12, that can be used in a keras model.

6.3.3 Keras model

With the embedding matrix in place (see subsection 6.3.2), we can use keras functional or sequential model API to construct our models. We've opted for the sequential model API, which we will demonstrate in this section, but both model APIs are perfectly viable. We have implemented several models in accordance with section 5.7, and since they are all implemented the same way, only differing in the types and amount of layers we will only describe the code for the *CNN-RNN* model that performed the best in our tests (see subsection 7.3.3).

The code for *CNN-RNN* is shown in Code fragment 6.13 where we start by initialising the sequential model, and adding the appropriate layers one by one. The first layer we add, lines 2-6, is the embedding layer, that serves to transform our input vector to an array of input vectors, which we covered in subsection 6.3.2. Then we add the CNN layers, the first of which can be seen in lines 8-12. We add several more layers in the same way, ending with a dropout layer, line 14, a GRU layer with 70 units, line 15.

To complete the model we add a dense layer, line 16, that aims to reduce the output to 1 unit to which the sigmoid activation function can be applied, line 17. With the layers added, we can then compile the model with the appropriate loss, optimiser and metric functions, such that we can start fitting the model, lines 19-21.

Code fragment 6.13: Constructing the *CNN-RNN* model using keras sequential model API

```
1  model = Sequential()
2  model.add(Embedding(embedding_matrix.shape[0],
3                      embedding_matrix.shape[1],
4                      weights=[embedding_matrix],
5                      input_length=100,
6                      trainable=False))
7  # First part
8  model.add(Conv1D(filters,
9                   kernel_size,
10                  padding='valid',
11                  activation='relu',
12                  strides=1))
13       ⋮ <- Additional layers
14 model.add(Dropout(0.5))
15 model.add(GRU(rnn_output_size))
16 model.add(Dense(1))
17 model.add(Activation('sigmoid'))
18
19 model.compile(loss='binary_crossentropy',
20               optimizer='adam',
21               metrics=['accuracy'])
```

6.4 Extending sentiment analysis model with negations

This section covers our extension to the initial SA design seen in section 5.7. Our extension serves to guide the sentiment classification by introducing a guiding feature, namely whether or not a word has been negated in a sentence or not.

The inspiration of this extension comes from our previous project where we built a neural network model, consisting of a RNN and an autoencoder to decide who the driver was for a given trajectory [7]. In this project we used a random forest, with our own features, where we found that starting and ending location was significant. We could therefore increase our neural network accuracy by including the starting and ending locations. This lead to the thought that adding our own features to the word vectors could help the network identify positive and negative documents.

In a sentence a word might be prefaced with a negation indicator. Our thought is that by explicitly identifying which words are negated, the model would be better at extracting the features representing a positive or negative document. This will be done by appending a 0 or 1 to the vector representation if the word is negative or positive, respectively.

6.5 Keyword extraction

Keyword extraction has been implemented using both approaches proposed in section 5.8. This section cover their implementation. Both approaches are implemented in Python, and the KG approach is implemented using the NetworkX library for building and querying the KG (see subsection 5.4.7).

The preprocessing phase will not be covered in code because of its simplicity and domain specific implementations. An algorithmic description of the preprocessing can be seen in Algorithm 1.

6.5.1 TFIDF

The process of extracting keywords using TFIDF is described formally in subsection 5.8.1. The implementation in Code fragment 6.14 is written in accordance with the algorithmic approach seen in Algorithm 2.

First, we calculate TFIDF for a single word in a single document with the code in Code fragment 6.14 which is implemented in accordance with the technical definitions in subsection A.4.1.

Code fragment 6.14: Code snippet of the Python functions responsible for calculating TFIDF for a single word in a single document

```python
# Term Frequency (TF)
def tf(t, d):
    freq = 0
    max_freq = 0
    for t1 in d:
        if t1 == t:
            freq += 1

        # Is this the largest seen so far?
        if freq > max_freq:
            max_freq = freq

    return freq / max_freq

# Inverse Document Frequency (IDF)
def idf(t, D):
    docs_with_t = [d for d in D if t in d]
    num_docs = len(D)

    # Prevent division by zero
    return math.log(len(D) / (1 + len(docs_with_t)))

# Term Frequency-Inverse Document Frequency
def tfidf(t, d, D):
    return tf(t, d) * idf(t, D)
```

Using the `tfidf()` function in Code fragment 6.14, the code in Code fragment 6.15 implements the function responsible for extracting *n* keywords from a corpus. The `tfidf()` function is called on line 15 when the function is filling the cells of the TFIDF score matrix with TFIDF scores. The function in Code fragment 6.15 is implemented in accordance with the algorithmic approach seen in Algorithm 2. First, the function finds all unique tokens in the corpus on line 3. Then, the TFIDF matrix is prepared by creating an array of dictionaries; the keys of the dictionaries being the unique tokens, and the value of each key being 0. A separate dictionary is declared on line 11 in order to keep track of extracted keywords and their frequency.

The cells of the TFIDF matrix are filled with TFIDF scores on line 15. As seen in the code, the TFIDF score is calculated for every word in every document.

After filling out the matrix with TFIDF scores, the matrix is processed in document-wise fashion. For each document, or `token_stream`, the corresponding dictionary of words and TFIDF scores are extracted in line 19. The list is sorted by the TFIDF score, reversed so the highest valued elements are at the head of the list, and we update the keyword/frequency dictionary with the top-*n* words on lines $23-27$.

Finally, the keywords/frequency dictionary is sorted, reversed, and the top-*n* words are returned on lines $30-33$.

Code fragment 6.15: Code snippet of the Python function responsible for extracting *n* keywords from a corpus given as `token_streams`.

```python
def keywords_with_tfidf(n, token_streams):
    # Unique words
    unique_tokens = set([token for token_stream in token_streams for token in token_stream])

    # Construct TFIDF matrix
    tfidf_map = []
    for _ in token_streams:
        tfidf_map.append({token : 0 for token in unique_tokens})

    # Store keyword frequencies
    keywords = {}
    for i, token_stream in enumerate(token_streams):
        for token in token_stream:
            # Calculate TFIDF and store in matrix
            tfidf_map[i][token] = tfidf(token, token_stream, token_streams)

    for i, token_stream in enumerate(token_streams):
        # For every document, get the word scores
        scores = [(tfidf_map[i][token], token) for token in unique_tokens]
        scores = sorted(scores)
        scores.reverse()
        # Take the top-n words and update the keyword frequency map
        for (score, word) in scores[:n]:
            if word not in keywords:
                keywords[word] = 1
            else:
                keywords[word] += 1

    # Store the keywords/frequency map on the frequency of the words
    keywords = sorted(keywords.items(), key=lambda kv: kv[1])
    keywords.reverse()
    keywords = [word for (word, freq) in keywords]
    # Return the top-n keywords
    return keywords[:n]
```

6.5.2 Graph building

As mentioned in subsection 5.4.7 we use NetworkX for building and querying the keyword graph. This process is implemented as seen in Code fragment 6.16. As can be seen in Algorithm 3, the implementation is consistent with the algorithmic approach devised in the design of the graph building process. There are, however, small differences in that we do not check for the existence of a node $n \in N$ at every iteration. Instead, we instantiate the graph nodes from the token set received as input. Every token stream is processed in sequence and is split up into k-shingles. Then, every shingle is processed where an edge is created between the first token in the shingle and its successors within the shingle, effectively conducting k-neighbouring edging as described in section 7.4.2. We have specifically chosen to conduct 4-neighbour edging. The reason behind this is both related to performance and effectiveness. Naturally, the more edges we introduce in the graph, the longer the closeness calculation will take as the task of finding the shortest path increases in complexity. Unrelated to the issue of pure computational performance, preliminary tuning of k revealed that, for a small number of test SMPs, the keywords did not change significantly with larger k's, and remained sufficiently descriptive.

After the edges have been created, we check the first part of the KG configuration; a KG can use three different weighting strategies and two different centrality measures. Here, we check for which weighting strategy to use, all of which are described in subsection 5.8.3.

First, we check whether or not we should conduct weighting. If not, all edges have automatically been assigned the same value. If we should conduct edging, we continue to line 23 where we start calculating the weights between nodes `n1` and `n2`. Both the absolute frequency difference and the inverted frequency difference is calculated, and we check once again which approach to take, after which we store the weight in a edge-to-node dictionary and update the sum of weights for this node.

After calculating the raw weights, we assign the weights to the corresponding edges after normalising the weights on lines 38 – 42.

Code fragment 6.16: Code snippet of the Python function responsible for building a keyword graph given a token set and token stream.

```
 1 def build_graph(self, weighting=None):
 2     graph = nx.Graph()
 3     graph.add_nodes_from(self.token_set)
 4 
 5     for token_stream in self.token_streams:
 6         # Create shingles for conducting k-neighbour-edging
 7         shingles = [token_stream[i : i + 5] for i in range(0, len(token_stream) - 4)]
 8 
 9         # Conducting 4-neighbour-edging. For every token, we add
10         # an edge between it and each the 3 following words.
11         # Equal weighting, so all edges have the same weight.
12         for shingle in shingles:
13             first_token = shingle[0]
14             for neighbour in shingle[1:]:
15                 graph.add_edge(first_token, neighbour)
16 
17         if weighting:
18             # Weight the edges of the graph
19             for n1 in graph.nodes():
20                 node_weights = {}
21                 sum_weights = 0
22                 for n2 in graph.nodes():
23                     if not n1 == n2 and graph.has_edge(n1, n2):
24                         #
25                         v1 = graph.node[n1]['freq']
26                         v2 = graph.node[n2]['freq']
27                         diff = abs(v1 - v2) # Absolute freq difference
28                         inv  = 1 / diff    # Inverted freq difference
29 
30                         if weighting == 'abs':
31                             node_weights[n2] = diff
32                             sum_weights += diff
33                         else:
34                             node_weights[n2] = inv
35                             sum_weights += inv
36 
37                 # Normalise the weights, then update edges
38                 for n2, weight in node_weights.items():
39                     if sum_weights == 0:
40                         graph.add_edge(n1, n2, weight=1)
41                     else:
42                         graph.add_edge(n1, n2, weight=weight / sum_weights)
43 
44     return graph
```

6.5.3 Extracting keywords from centrality measures

As described in section 7.4.2, we extract top-n keywords based on the centrality of their assigned nodes in the graph. We include both closeness and PageRank as possible centrality measures. This process is implemented as seen in Code fragment 6.17. As can be seen in Algorithm 4, the implementation is consistent with the designed approach in section 7.4.2.

Given a graph as build with the `build_graph()` method in Code fragment 6.16, we immediately check which centrality measure to use on lines 5 and 8. The result of the centrality calculations is a list of word-score pairs. The returned list is sorted, reversed, and the top-*n* elements are returned as the keywords for this corpus.

Code fragment 6.17: Code snippet of the Python function responsible for extracting the top-n keywords from a keyword graph.

```
1  def extract_n_keywords(self, n = 5, centrality='pagerank'):
2       # Build the graph
3       graph = self.build_graph()
4       scores = []
5
6       if centrality == 'pagerank':
7           # Calculate pagerank (receive a list of pairs (node, rank))
8           scores = nx.pagerank(graph)
9       else:
10          # Calculate closeness
11          scores = nx.closeness(graph)
12
13      # Sort the scores
14      closeness_scores = sorted(closeness_scores.items(), key=operator.itemgetter(1))
15      closeness_scores.reverse()
16
17      # Return the top-n words
18      return [word for (word, closeness) in closeness_scores[:n]]
```

Chapter 7: Test and evaluation

In this chapter we will describe the process we have gone through in order to test our overall systems and essential components in isolation in order to achieve different goals ranging from throughput tests and detecting bottlenecks. We also conduct tests to see how well our individual components work together in a combined system.

We perform throughput testing of the gateway and statistics services in section 7.1.3, and our usability test is detailed in section 7.2. In section 7.4 and section 7.3 we evaluate the performance of various KE and SA approaches respectively. Finally, unit and integration testing of our application is covered in section 7.5.

7.1 Web-application throughput

In this section we will perform a systematic test of the response time of some of our services under increasing loads. In particular, we will be testing the response time of the gateway service in subsection 7.1.3 and the statistics service in subsection 7.1.2. For each service, we will be testing them with a single node and with two load balanced nodes. In either scenario, the quantity of data in the databases will be similar, though it is subject to change as a result of the simulation.

7.1.1 Throughput test setup

In order to test the throughput of these services, we will be using the load testing framework Locust[1], wherein various tasks from the perspective of the service consumers have been implemented. We will not be testing the SA and KE models in this section, as these are be covered in section 7.4 and section 7.3. The purpose of these tests are not meant to be a binary classification on whether the system is scalable or not; instead, we are using the test to determine how scaling out requests to multiple nodes affects our performance.

Due to limitations on the resources we have available to us, we cannot test our system under optimal circumstances as we only have one dedicated server which hosts the majority of our services. For tests with more than one node, we will be using our own laptops to deploy the service. Under ideal circumstances, we would have one node per service and per database. Despite this limitation, we hope that the test can show how using load balancing affects the performance. The system

[1]See: `https://locust.io/`.

specifications for the dedicated system are available in section 2.4. The secondary system to be used in the load balancing test has as Intel i7-5600U Dual core CPU @ 3.2Ghz as well as 8 GB RAM.

7.1.2 Statistics service

In the context of the statistics service, growth entails the increasing amount of snapshots saved over time. In the worst case, our system saves one snapshot for every active synonym in our system at some specified time span (e.g. every hour). In our simulated scenario, we have saved 4.3 million snapshots, corresponding to a snapshot being saved every hour for 500 synonyms over the course of one year. The challenge is not only a space scalability problem, however, as the system can be exposed to an increasing amount of statistics requests which requires processing power to handle. This test will focus on a load test, where an increasing amount of consumers request statistics from different synonyms concurrently. The system to be used in the load balancing test was described in subsection 7.1.1.

Preliminary optimisations

Prior to testing the throughput of the statistics system, we ran some optimisations on the snapshot relation in our database using the `EXPLAIN ANALYZE` statement in PostgreSQL. We found that the execution time for querying snapshots was quite high given as the DBMS was performing a sequential scan for each query. In order to speed up the queries, an index was added for the synonym column of the snapshot relation.

Experimental setup

The experimental setup for our statistics load experiment can be seen in Table 7.1. The consumer hatch rate is the frequency of new consumers starting every second. The minimum and maximum action delays refer to the minimum and maximum amount of time a single consumer can wait before performing a request to the node. In each scenario, we will run the test until 1300 concurrent consumers are reached.

	Single node	**Load balanced**
Amount of nodes	1	2
Consumers hatch rate	10	
Minimum action delay	5.0 sec.	
Maximum action delay	10.0 sec.	

Table 7.1: Experimental setup for the statistics load experiment.

In terms of measurements, we will be recording the throughput (in terms of Requests Per Second (RPS)), median response time, median DBMS query and average processing time. This should help us determine how the increasing load affects both the database and the service itself.

Activities

The statistics service currently has two consumable endpoints:

1. Overview: Given a date range, granularity and a set of synonyms, the service computes statistics over this period.
2. Brand: Given a set of synonyms and a time span, the service computes the average sentiment and sentiment trend from the current time spanning back the specified time span.

In this test, we will only be focusing on the overview endpoint, since the brand endpoint is rate limited. The requests will be made using random date ranges, granularities and synonym sets, in order to be representative of the possible diversity of queries. The granularity can be hour, half day, day or week. We have excluded certain queries, such as the ability to query over more than a day with an hourly granularity.

Hypotheses

We expect to see that the single node scenario will have significant spikes in response time before the load balanced scenario. We expect that the single node scenario will have a lower base response time, since the database is hosted on the same machine and since there is no overhead of load balancing. Finally, we expect the load balanced scenario to have a higher base response time, both due to the aforementioned reasons and the weaker specifications of the secondary node.

Results

The results of the scenario with a single node can be seen in Figure 7.1. In this scenario, the response time remains stable around a median value of 100 milliseconds, until 450 concurrent users are reached where the response time begins increasing. At the same amount of concurrent users, the throughput begins capping around 65 RPS.

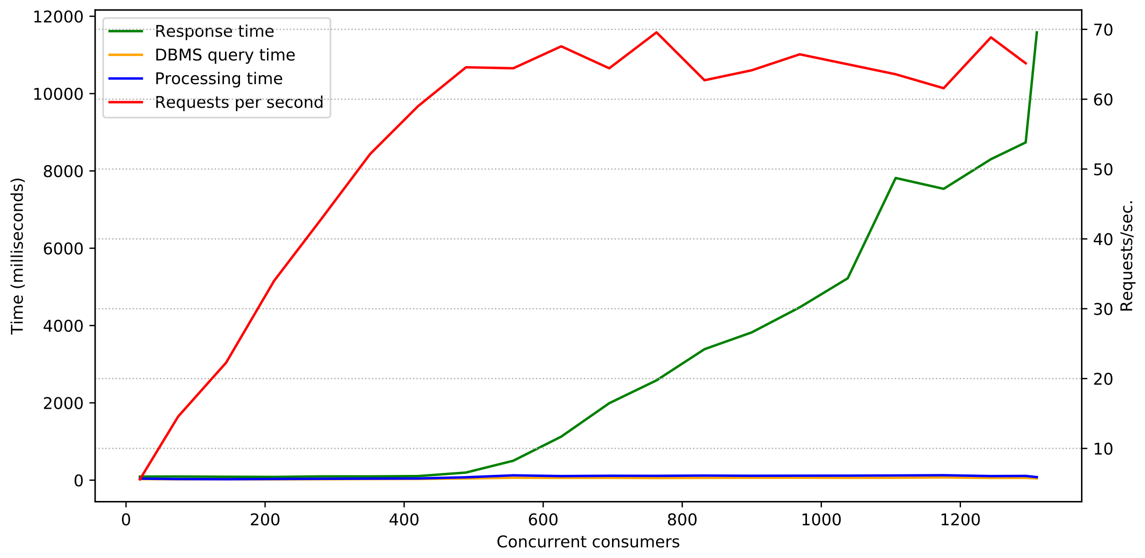


Figure 7.1: Results for the statistics throughput test with a single node.

Due to disparities in the computational power of our nodes, we experimented with different weightings of the nodes in the load balancer. Weightings mean that that given two nodes n_1 and n_2 with weights w_1 and w_2, the load balancer will choose n_1 w_1 times whenever it has chosen n_2 w_2 times. We found the optimal weighting to be 2:1 such that the main node receives twice as many requests as the auxiliary node. Different weightings and their respective RPS can be seen in Table 7.2, including weightings where only the primary or only the secondary node are used.

Main node weight	**Auxiliary node weight**	**RPS**
1	1	75.0
2	**1**	**100.2**
3	1	86.5
1	3	48.4
1	2	51.5
1	0	65.0
0	1	35.0

Table 7.2: Weightings for different nodes with their corresponding RPS.

The result of the scenario with two nodes can be seen in Figure 7.2. In this scenario, the median value of the response time is initially stable at around 100 milliseconds. At around 750 concurrent consumers, the response time steadily increases and similarly to the other scenario, the throughput caps out at, however this time around 90 to 100 RPS. Even under the increased load, neither the processing time or DBMS query time are affected significantly, indicating that there is still room for scaling out the service without having to scale out the DBMS. Note that following this experiments we carried it out again with automatic load balancing which showed similar results to those in the 2:1 weighting scenario.

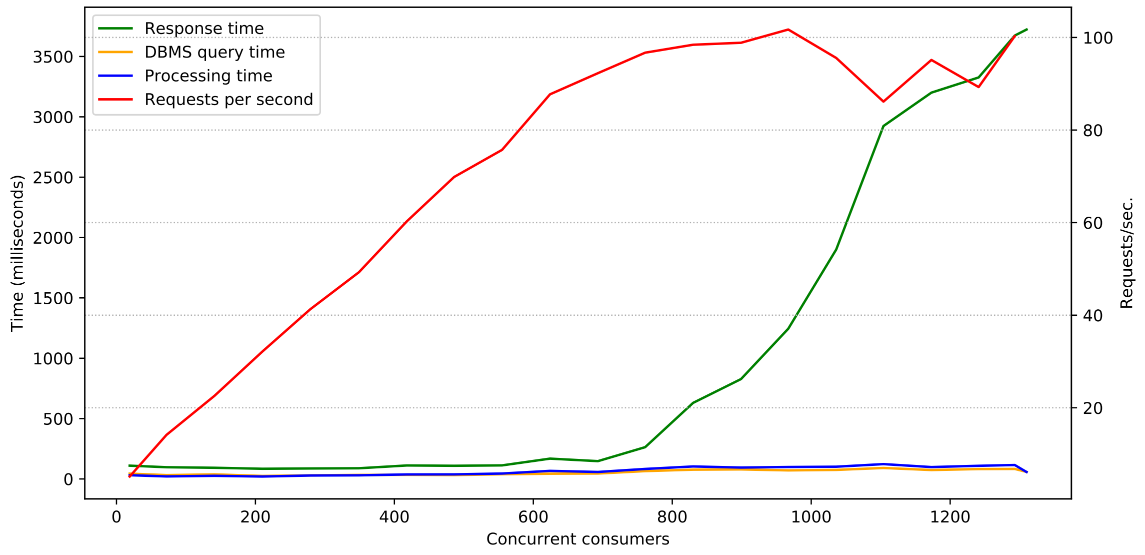


Figure 7.2: Results for the statistics throughput test with two nodes using a 2:1 node weighting for load balancing.

Evaluation

As we had hypothesised, the response time of the scenario with multiple nodes remained stable under a higher load than the scenario with the single node. The increase in throughput when adding a secondary node was approximately equal to the throughput of the secondary node on its own, which suggests a minimal overhead with scaling out. As a result of the lower response times, having multiple nodes yielded an increased throughput. In either case, the DBMS query time seemed unaffected by the increased load. Based on this we can conclude that there is still potential to scale the statistics service out to more services.

Another bottleneck in the current choice of database that is unidentifiable from this test is that of space scalability. As the number of snapshots will keep growing with no user interaction, 1 GB of storage being capable of storing a years worth of snapshots for 250 synonyms, the host machine will eventually run out of space even when the system is unused. A possible solution to this issue is to use sharding (e.g. sharding the snapshots by year), which would allow different nodes to host a number of shards, however this is not supported by our current database system.

7.1.3 Gateway service

In this section we will shift our focus to testing the gateway service. As a brief reminder, this service is the only service directly accessible by consumers of our system. It handles CRUD operations relating to accounts, brands and synonyms. Additionally, it acts as a message broker to different services, e.g. by forwarding requests to the statistics service.

The database assigned to this test contains 100.000 accounts and 250.000 brands, which have been randomly assigned to the accounts with one synonym each.

Experimental setup

We will be following an experimental setup similar to the one described in subsection 7.1.2, however instead of recording processing time and DBMS query time, we will be recording the response time for each service call, which rougly corresponds to DBMS query times. Additionally the hatch rate has been reduced to 5 consumers per second better reflect the behaviour of consumers of the service.

Activities

Given that the gateway service is the only service available to consumers, many CRUD operations are available for use.

1. Account: Consumers are able to create an account.
2. Authorisation: A user is able to log in and log out.
3. Brand: Consumers can create brands, update them and delete them.
4. Synonym: Consumers can add and remove associations between a synonym and a brand.
5. Statistics: Consumers can requests statistics for a brand given some date range and a granularity.

In this test we will ignore the statistics activity as the underlying service has already been tested in subsection 7.1.2. Since a number of accounts have already been created for the purpose of this test, the account creation activity will be invoked less frequently than the other activities. During the initial part of the test, all consumers will be performing logins, but will randomly log out and log back in during the test.

Hypotheses

Our hypotheses are similar to the ones from subsection 7.1.2, i.e. we expect to see that the response time will increase faster in the single node scenario than in the load balanced scenario. Furthermore, we expect to see that the authorisation and account endpoints will have a higher response time than the other endpoints, as both these endpoints hash account passwords over several rounds, requiring significant computational power. Because of the cost on these endpoints, we expect to see that there is a spike in response time initially as consumers hatch and log in.

Results

Due to the aforementioned disparity in computational power, we chose the same 2:1 weighting of the two nodes in our load balancer as in subsection 7.1.2. The result from the single node scenario can be seen in Figure 7.3. In this scenario the response time remains table around 80 milliseconds until 600 consumers are reached. At the same points, the throughput caps out at around 85 RPS. Although not visible from the plot, our service response times show that account creation and account logins are slow, averaging around 500 milliseconds per invocation, which is due to the aforementioned hashing of passwords. On the other hand, it is evident from the initial spike in response time where many accounts perform logins, after which the response time stabilises.

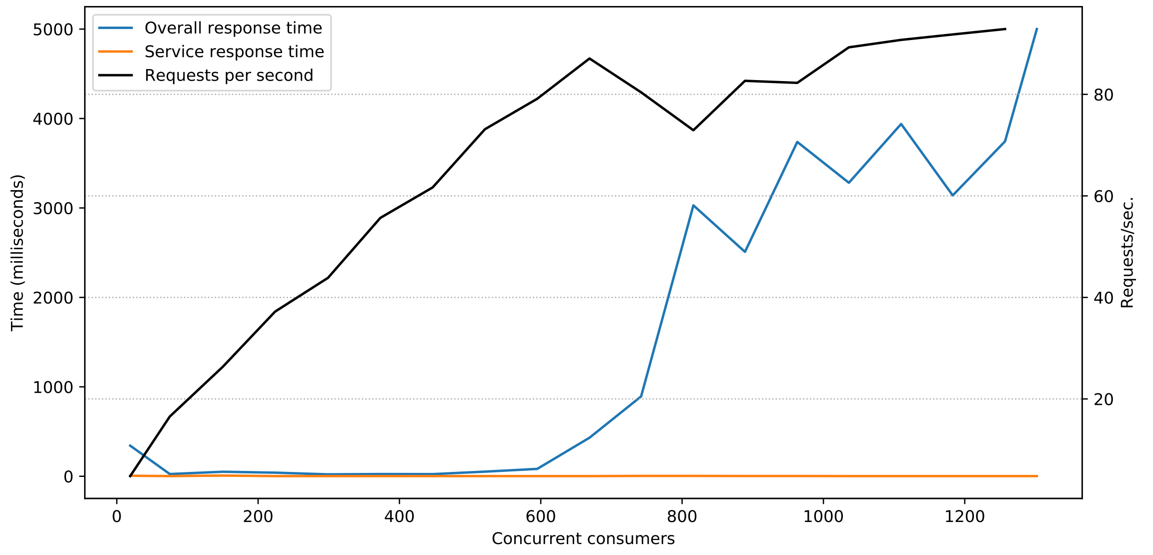


Figure 7.3: Results for the gateway throughput test with a single node.

The results from the multiple node scenario can be seen in Figure 7.4. As hypothesised, the response time remains stable for a longer time at around 100 milliseconds until 950 consumers are reached. At that point, the throughput caps out at 125 RPS which is significantly higher than the throughput seen in the single node scenario. Even with the increased throughput, the stability of the service response time indicates that the DBMS is unaffected thus far.

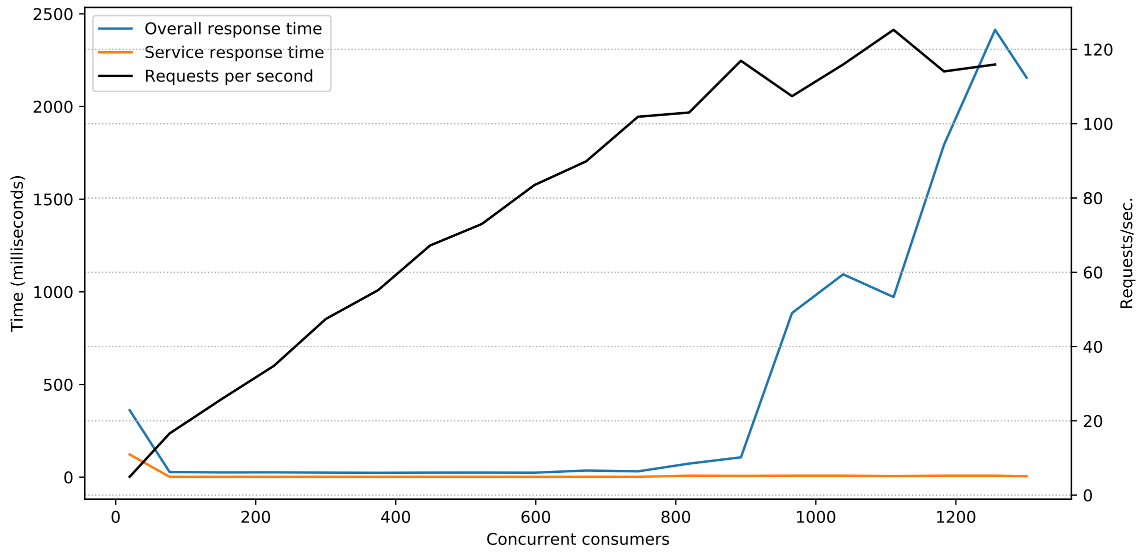


Figure 7.4: Results for the gateway throughput test with multiple nodes.

Evaluation

As observed in the results of the previous tests, the response time of the scenario with multiple nodes yielded a roughly 50% higher throughput than having a single node. The stability of the service response times indicates that the gateway service can still be scaled out to more nodes without having to do data replication or sharding on our DBMS. We could improve our throughput by using another hash algorithm or using less rounds, but it would be less secure.

7.2 Usability test

In this section we will look at the usability of our system through the eyes of a test subject following the assignment in appendix C.

Test

For the first assignment the subject had to create a new user account, starting from the log in component the subject was first presented with. The subject quickly identified the hyperlink with the text "Register?", clicked it and got to the registration component. The subject filled in his credentials, clicked the button with "Register" and was taken back to the log in component.

For the next assignment the subject had to log in using the newly created user account. The subject filled in his credentials, clicked the button with "Login" written on it.

For the third and fourth assignment the subject had to add a brand. The subject quickly found the button with the plus sign and clicked it, saying that was the only thing that made sense. The subject then identified the input field, wrote the name of the brand and clicked the create button.

For the fifth assignment the subject had to add a synonym to one of the newly created brands. The subject looked around for a bit, trying to click the button with three dots to the right of the brand in the table. After the subject found that there was no option to add a synonym there, the subject tried to click the name of the brand to go to the brand page, where the subject identified the synonyms component after scrolling down on the page, which the subject mentioned to be a bit inconvenient. The subject then wrote the name of the synonym in the input field that had the placeholder "Add synonym" and pressed enter, which added the synonym to the table of synonyms.

For the sixth assignment the subject had to find the top keywords for a brand at a given time. The subject had previously identified the graphs on the brand page and so quickly thought to return to the brand page. The subject looked for a graph that was specific to keywords, unable to find one the subject looked at the first graph to try and find the keywords. After a while and still no luck, the subject turned to us for help and we showed him that it was possible to hover a single point on a line, depicting the sentiment over time for a synonym, in the line chart to find keywords for that specific synonym. The subject noted that the most popular keywords which included 'staff' and 'UI' made sense, as the subject was actually an avid follower of the brand used for the example and said the brand had just hired some new UI staff members and sparked a debate in the community.

For the seventh assignment the subject had to identify whether there were more positive or negative posts over the past week. The subject quickly identified the bar chart and found that the majority of posts were negative.

For the eighth assignment the subject had to alter the settings for one of the brand pages to reduce the amount of statistics data depicted by the graphs and set the granularity of the data to hourly. The subject quickly identified the cog wheel in the top bar of the page, clicked it and was presented with the settings bar for the page. The subject changed the settings and clicked apply.

For the ninth assignment the subject had to determine the sentiment peak value for a brand. The subject identified the need to look at the line chart and quickly found the peak value by looking at the graphs data.

Evaluation

From watching the test subject, the system seemed to be very intuitive as there were few times where the subject was held up on a problem. The few times where the subject was held up helped us identify the following issues:

- The subject intuitively figured it would be possible to create a synonym for a brand using the row actions (the button with the three dots) in the table of brands in the brands page instead of having to go to the brand page.
- The synonyms component was difficult to find at the bottom of the brand page, as it was placed below both graphs and thus pushed out of view.
- The subject did not find it intuitive to hover over a point on the line chart to read brand statistics at the given point in time.

7.3 Performance comparison of sentiment analysis models

This section will show the result of our tests SA models described in subsection 5.7.4 and our evaluation of these. subsection 7.3.1 describes how we test our models, what our focus is and why, followed by subsection 7.3.4 that describes our testing results and our evaluation of those.

7.3.1 Testing focus

We have three focus areas when testing our models. We focus on the accuracy of our model, where the most important is its F-measure, as it describes how well the model is on both positive and negative labelled documents. We can say that the accuracy on both positive and negative predictions are equally good if the F-measure is equal to the accuracy. We can say this because our test set is balanced. The accuracy of positive and negative predictions is important as we want to correctly classify as many documents as possible, making the keywords extracted more relevant. Though given enough documents it should be possible for the KE algorithm to extract the correct keywords even for a model with slightly lower accuracy. The loss of a model is also relevant as we want the model to be as certain of its prediction as possible. If a models loss is high, it means that many predictions are centered around 0.5 resulting in a daily graph with few changes, even when the sentiment changes for a brand. The last is the throughput of the models as we want to be able to as many document in a given time period as possible.

7.3.2 Experimental setup

We ran all tests on a computer with an Intel i7-5600U CPU @ 3.2Ghz and 8 GB RAM. The models were trained using the adam optimiser, that automatically updates the learning rate. The CNNs had the same number of feature maps and receptive field size, during testing. Both RNN models had the same number of hidden units. We use 320000 documents from the sentiment140 dataset, where we assume that this set is representative for the documents we will get from crawling the different websites. A model that achieves good results on this subset of sentiment140 would therefore also achieve good results when running as a service in our system.

The throughput of the model is how long it takes a model to process and predict the sentiment for the 320000 documents. The environment was not completely sterile, meaning some background processes might interfere with the throughput test, though this would not have a big enough impact to invalidate the test or distort the general performance insight given. All models was tested on the same test set, meaning a model might just good only for this dataset and not generally.

7.3.3 Model results

The results of our models are in Table 7.3. The table shows the results of our different models where the CNN & GRU a model where we run the best performing CNN and best performing RNN in parallel. For each output of the two models we choose the prediction with the most confidence in its prediction. This model was therefore run using the *ConvPool-CNN* and GRU models. The *CNN-RNN* model is the model with a CNN model followed by the the best RNN.

	LSTM	GRU	LSTM & GRU	Strided-CNN	ConvPool-CNN	All-CNN	CNN & RNN	CNN-RNN	Naive Bayes
Acc	0.802	0.804	0.806	0.734	0.810	0.741	0.815	**0.820**	0.759
Recall	0.777	0.803	0.792	0.754	0.760	0.792	0.789	**0.827**	0.734
Precision	0.818	0.805	0.814	0.726	**0.845**	0.719	0.833	0.817	0.773
F1	0.797	0.804	0.803	0.740	0.800	0.754	0.810	**0.822**	0.753
Loss	0.426	0.423	0.423	0.524	0.416	0.514	0.413	**0.399**	0.519
Throughput	919.30	1039.13	500.48	1386.54	603.56	722.92	376.92	548.78	**103896.10**

Table 7.3: A table containing the results of our models, with the best result highlighted for each measurement. The different accuracy measurements are described in section A.5. The loss is the cross entropy. Throughput is documents per second.

We tested the best performing NN models with with negated words, as described in section 6.4. The results are shown in Table 7.4.

	GRU-NEG	ConvPool-CNN-NEG	CNN-RNN-NEG	Naive Bayes-NEG
Acc	0.801	0.810	**0.817**	0.742
Recall	**0.828**	0.754	0.812	0.777
Precision	0.787	**0.849**	0.821	0.726
F1	0.807	0.799	**0.816**	0.751
Loss	0.426	0.412	**0.403**	0.718
Throughput	923.73	586.07	551.83	**94955.49**

Table 7.4: A table containing the models trained on negated words. The different accuracy measurements are described in section A.5. The loss is the cross entropy. Throughput is documents per second.

7.3.4 Evaluation

The *CNN-GRU* in Table 7.3 performs best in multiple measurements. And seems to extract the dependencies of sentences in the documents. The Strided and All CNN models perform worse than the NB model, meaning they perform worse than a model that does not take temporal or spatial information into account. These to models therefore are not interesting to look at. Both *Strided-CNN* and *All-CNN* have strides, meaning that the effectively skips some units of its output, meaning some dependencies between words might have been skipped.

The *CNN-RNN* is almost 200 times slower than the NB model. Though we do not deem this as important as the SA node in our system is stateless, meaning it is possible to scale out this component. Furthermore, even on limited hardware, it reached 549 documents per second, which can be improved by scaling up. In section 7.1.3, we use 500 synonyms to test our system. We do not think that there will be more than 1000 posts per synonym per hour in average, which means that our SA program should be able to classify 139 post per second, to handle this. The *CNN-RNN* could easily handle this load and its F-measure is better than NB.

The models trained on negated words, described in section 6.4, does not perform better, but a little worse. Our theory as to why it does not increase the accuracy is that it disrupts the features of the word2vec embedding. The distance between similar words might be increased, and decreased for dissimilar words, meaning even though we add information that should make it easier for the model to recognise features, it also removes some information.

Both combinations of CNN and RNN performs better than when they operate individually. Using both RNN in combinations does not increase performance, but actually lowers the F-measure a little. These two observations indicates that CNN and RNN extracts different features as described in section A.1 and section A.2.

If a customer thought that negative posts were important, we could use the *ConvPool-CNN* instead of *CNN-RNN*. Having a high precision means its negative recall is high, and therefore classify negative documents correctly more frequently than positive documents. As the *ConvPool-CNN* and *ConvPool-CNN-NEG* has higher precision than *CNN-RNN* it would be better for this purpose. As we cannot say anything about a customers demands, we will use the model that performs best on both positive and negative documents, which is the *CNN-RNN*.

The *CNN-RNN* also have the lowest loss, which means its confidence on its prediction is higher than the other models, as described in subsection 7.3.1. Since the *CNN-RNN* performs best in accuracy, F-measure and loss, it is the model that is used in our system.

7.4 Performance comparison of keyword extraction approaches

This section compares the performance of the two proposed approaches for KE; TFIDF and KG. The technical specification of these approaches can be found in section 5.8. Since our system must both be scalable and load tolerant, it is likely that we must incur a trade-off between the quality of the extracted keywords and extraction time. In subsection 7.4.1, we compare the running times of TFIDF, $KG|C|X$ and $KG|P|X$ over a growing number of documents, where $KG|C|X$ represents that we take the average running time for $KG|C$ using all three weighting strategies. In subsection 7.4.2, we compare the different configurations of KG proposed in subsection 5.8.3 in terms of their precision. Precision, as the number of correctly extracted keywords, is defined formally in section A.5.

Performance was measured over a set of 50 to 1000 documents from the test data set of Anette Hulth which was first used in [23]. Each sample is an abstract from a scientific paper that is associated with a set of correct keywords.

When referring to configurations of the KG based approaches, we use the same naming scheme as described in section 5.8.

7.4.1 Running time comparison

The results of the running time experiments can be seen in Figure 7.5. Keywords were extracted from 5, 10, 25, 50, 100, 200, 500, and 1000 documents of varying length. The running time was measured in seconds from when the algorithm starts to when it terminates.

As can be seen in Figure 7.5, TFIDF and $KG|P|X$ performed similarly with $KG|P|X$ being much faster as the number of documents increases. Data is missing from $KG|C|X$ after 200 documents since the closeness centrality calculation failed to terminate within five minutes, rendering it unusable for our project and its results irrelevant.

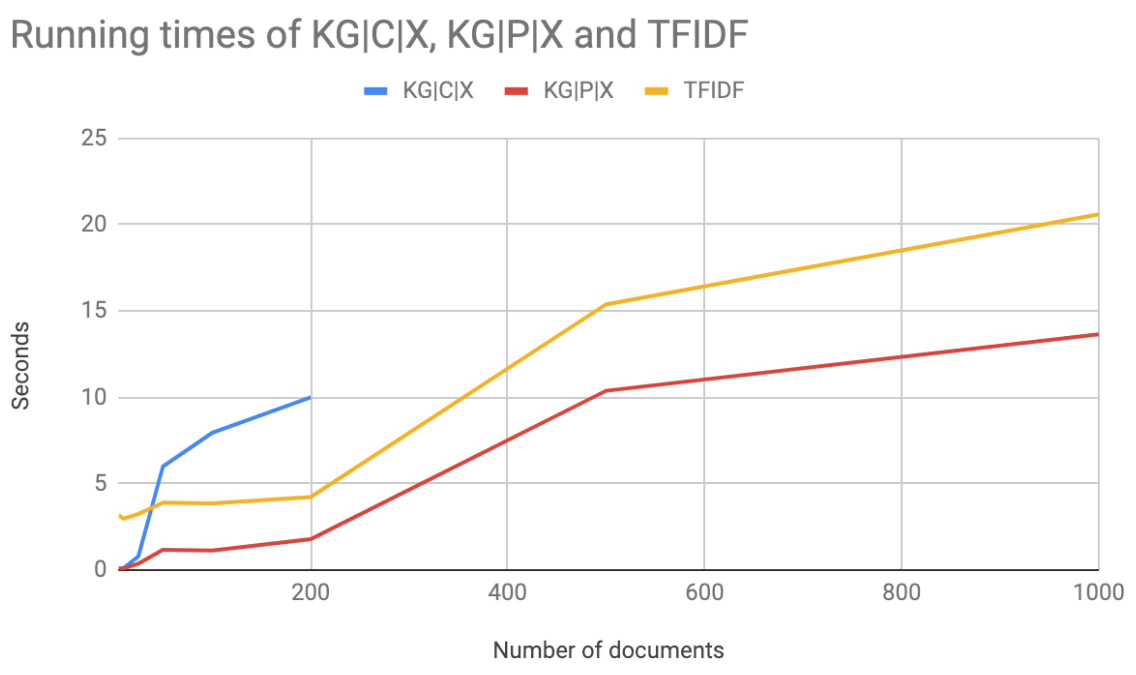


Figure 7.5: Comparison of running times between extracting keywords using TFIDF, $KG|C|X$ and $KG|P|X$. Keywords were extracted from 5, 10, 25, 50, 100, 200, 500, and 1000 documents of varying length as seen along the horizontal axis. The running time, as seen along the vertical axis, was measured in seconds from when the algorithm starts to when it terminates. Data is missing after 200 documents for $KG|C|X$ as it failed to terminate within five minutes.

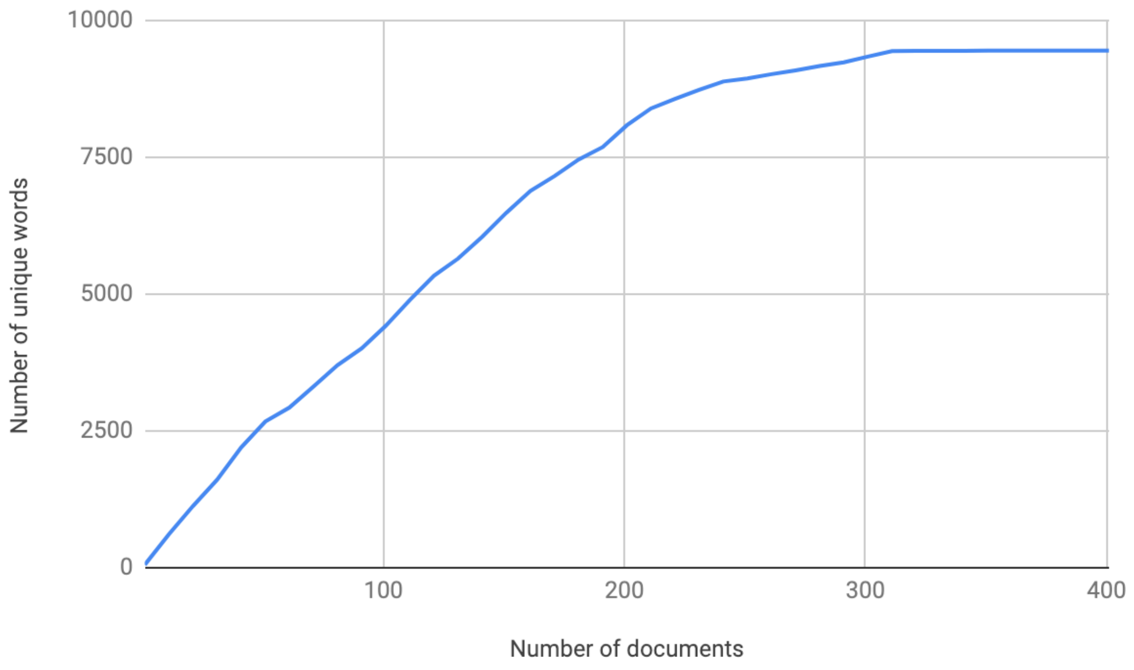


Figure 7.6: The total number of unique words as the number of documents increases.

Time complexity of KG|C|X

We expect that the failing termination of $KG|C|X$ is caused by the time complexity of the closeness centrality calculation. Regardless of the algorithm used for calculating shortest paths, it is required to calculate all shortest paths from every node to every other node. Using a simple implementation of Dijkstra's algorithm to calculate shortest paths, the time complexity of calculating closeness for every node in the graph has time complexity $O(N^3)$ where N is the number of nodes in the graph (the simple implementation of Dijkstra's algorithm has time complexity $O(N^2)$ [14, p. 595–601]). Recall that there are as many nodes as there are unique words in the corpus.

A simple experiment of counting the number of unique terms as the number of documents increase shows that, while the number of unique words quickly converge (around 300 documents in our case), the number still goes over ~ 9000 within 300 documents. This can be seen in Figure 7.6. Naturally, calculating closeness centrality for 9000 nodes is not a practical approach if we aim to be load tolerant.

Since we expect the number of documents to extract keywords from to far exceed that of ~ 300, we determine that the $KG|C|X$ approaches are not fit for this project.

Next, we compare TFIDF and $KG|P|X$ as they both succeeded to terminate as the number of documents grew.

7.4.2 Performance comparison of KG | P | X and TF-IDF

As discovered in subsection 7.4.1, $KG|C|X$ failed to terminate as the number of documents grew. As such, we will not evaluate its performance as the approach is not adequate for our purposes. This section directly compares the performance of TFIDF and $KG|P|X$ in terms of their precision, recall, and F-measure, all of which are defined in section A.5.

Note, however, that the goal for this experiment is to verify whether or not the words extracted from the graphs are indeed keywords, i.e. we are primarily interested in the precision. We extracted a constant (5) number of keywords for the corpus for which each containing document had assigned a varying number of keywords. The experiment was conducted using the same data set as in the running time experiment described in subsection 7.4.1. The results of this experiment can be seen in Figure 7.7 and Table 7.5.

We also extracted the same number of keywords with each individual sample text being the entire corpus and calculated precision, recall, and F-measure according to the number of keywords for that sample that were matched. The results of this experiment can be seen in Figure 7.8 and Table 7.6.

The results of the experiment can be seen in Figure 7.7 and Table 7.5.

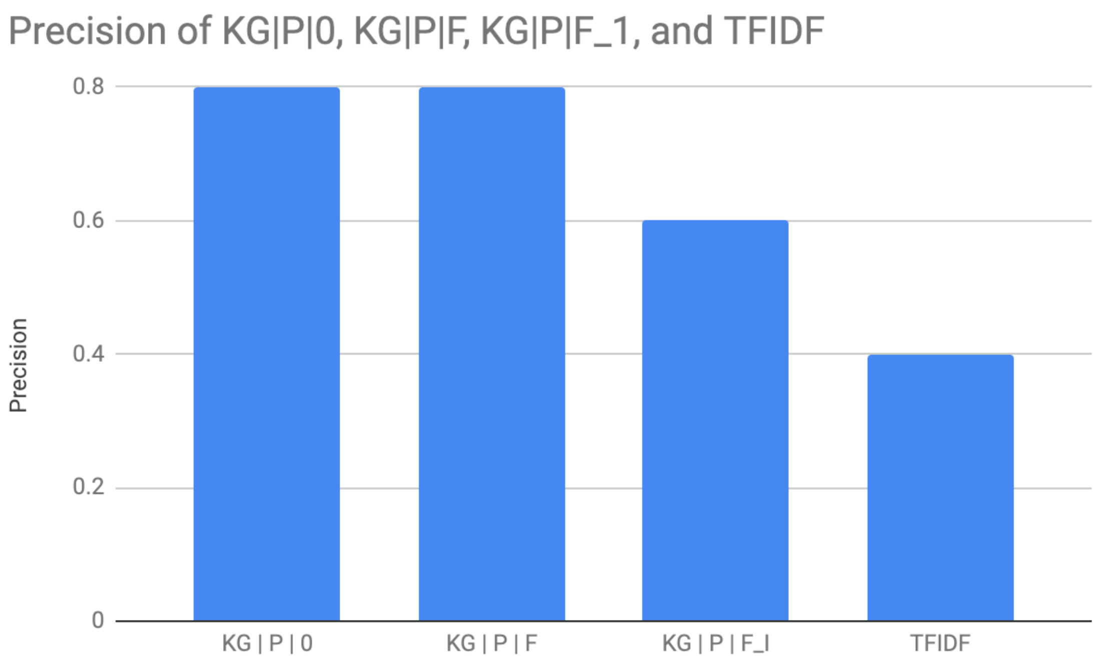


Figure 7.7: The precision score of all $KG|P$ configurations and TFIDF when extracting keywords over a corpus of 1000 documents.

As can be seen in Figure 7.7, the best performing approach is $KG|P|0$ with a precision of 0.80, meaning that only one of the keywords extracted from the corpus was not a keyword included in the keywords of the corpus. Similarly, after extracting keywords form single texts, the precision of $KG|P|0$ is 0.61, once again ranking the highest among the other approaches. Unfortunately, TFIDF

	$KG\|P\|0$	$KG\|P\|F$	$KG\|P\|F_I$	TFIDF
Precision	0.80	0.80	0.60	0.40
Recall	0.00	0.00	0.00	0.00
F-measure	0.00	0.00	0.00	0.00

Table 7.5: Performance comparison between the KG configurations that use PageRank as the centrality measure, and TFIDF after extracting keywords over a corpus of 1000 documents.

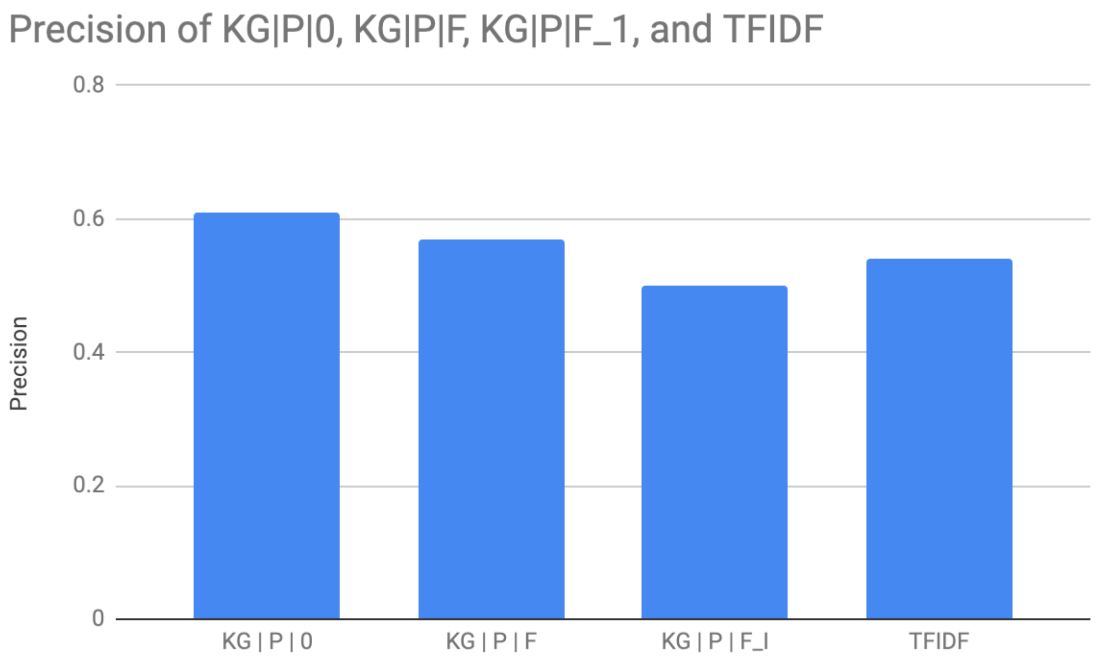


Figure 7.8: The precision score of all $KG|P$ configurations and TFIDF after extracting keywords from 1000 single-text corpuses.

	$KG\|P\|0$	$KG\|P\|F$	$KG\|P\|F_I$	TFIDF
Precision	0.61	0.57	0.50	0.54
Recall	0.07	0.07	0.06	0.07
F-measure	0.14	0.12	0.11	0.12

Table 7.6: Performance comparison between the KG configurations that use PageRank as the centrality measure, and TFIDF after extracting keywords from 1000 single-text corpuses.

falls short of the KG approach using PageRank with a precision of 0.4 and 0.54 for the experiments with a large corpus and with single-text corpuses, respectively. TFIDF extracts keywords based on the importance to the specific document. As such, given that the approach scores higher in the experiment with single-text corpuses as seen in Figure 7.8 implies that our method of extracting keywords over a large corpus using TFIDF is not optimal.

Both $KG|P|F$ and $KG|P|F_I$ perform worse than their identically weighted counterpart $KG|P|0$. It is important to note that the chosen weighting strategies F and F_I, as described in subsection 5.8.3, were chosen from our intuitive understanding of how a keyword could be determined. These results show that frequency difference, whether absolute or as inverse, are not features that can be used effectively with the PageRank algorithm.

Choice of approach

This section compares different approaches for KE both in terms of running time and precision. Seeing that $KG|C|X$ failed to terminate with large corpuses lead us to exclude the approach as a candidate for KE in our system. While TFIDF and $KG|P|X$ were all able to handle large corpuses, the $KG|P|0$ approach worked the best both in terms of running time compared to TFIDF and precision compared to all other approaches. As such, $KG|P|0$ should be used for conducting KE in our system.

7.5 Unit and integration tests

- you may add
es of test cases
eresting issues...

In this section, we present the code coverage and test status for all projects that make up our back-end. The following is a description of the different projects and the functionalities they implement. The results can be seen in Table 7.7. Note that the Angular client project, statistics API and sentiment analysis API are excluded as they currently have no tests.

Keyword Extraction Contains code files for conducting keyword extraction using KGs. Also contains code files implementing text pre-processing.
Keyword Extraction API Exposes endpoints to perform keyword extraction.
Information Retrieval Contains code files for our crawlers and for scheduling SA and KE.
Gateway Exposes endpoints to users such as statistics, authorisation and brand management.

Project	Coverage	Passing	Failing
`Keyword Extraction`	84%	100%	0%
`Keyword Extraction API`	98%	100%	0%
`Information Retrieval`	90%	30%	70%
`Gateway`	93%	100%	0%

Table 7.7: Code coverage, passing, and failing percentage for all unit and integration tests written for the associated project.

For the `Information Retrieval` project, we see that several tests are failing. While the tests were all passing earlier during the project, the failures are caused by Trustpilot changing the layout of their website, including the naming and structure of the HTML and CSS that make up their web-pages. This has caused the tests for the Trustpilot crawler to fail.

Chapter 8: Discussion

In this chapter we discuss and evaluate the choices made during our project design, implementation, and testing phase including the results we have achieved. In terms of requirements, both functional and non-functional, we argue to which extent these have been fulfilled.

8.1 System architecture

The requirements listed in chapter 4 require that our system architecture provides the ability to scale outwards as the amount of data and data processing increases. Furthermore, we require our system to be robust, i.e. it should be resistant to the failure of individual system components. In subsection 5.3.1 we designed our system architecture in accordance with the gateway-routing architectural design pattern, and all logical components of the architecture (the gateway, analysis, and statistics components) are implemented as self contained services that expose a number of endpoints defined by their underlying services, as described in section 6.1. First regarding robustness, splitting our system into services allows one component in the system, e.g. the analysis component, to fail without the rest of the system being affected. The analysis component can then be redeployed as a new service automatically. We originally designed the analysis component (see Figure 5.1) as one single component wherein the KE and SA modules are both services. In our implementation, KE and SA are implemented as self-contained services as well, and the analysis component makes use of them through APIs exposed by the corresponding services. There are several reasons behind this. First, if either of the KE or SA services fail, they can be redeployed. Furthermore, this allows for increased scalability.

The scalability of the system is directly supported by our component-oriented system architecture. In order to see this, we give an example with the analysis, KE, and SA components. Our current analysis module contains one scheduler that schedules information gathering from Reddit and Trustpilot using our web-crawlers (see Figure 5.3, subsection 5.3.4). If the system were to grow, we have the ability to simply add more components with schedulers and crawlers which would increase our information gathering throughput. Furthermore, and in relation to the notion of distributing complex computations as mentioned in section 8.5, the analysis component is not limited from making use of one KE and SA module. It is entirely possible for us to add additional services conducting KE and SA, the results of which can be combined within a single (or several) analysis component(s).

In the system architecture design, we introduce a load balancing component between the client and the gateway component. This is not used in our current project, but we did implement a load balancer for the throughput test described in subsection 7.1.1. The reason for not using a load balancer for our actual system is that we have only had a single development machine to work with, so a load balancer would not afford us any additional computing power as there are no additional machines to distribute the workload to. However, if we were to scale the system by adding nodes as has been done in the throughput test, a load balancer that takes advantage of both server capacity and locality could benefit the users of the system by having reduced response times. As our implementation supports the addition of a load balancer, the system is prepared for scaling by adding nodes.

The scalability of our architecture is further enabled by our system being implemented according to the Representational State Transfer (REST) architectural style, making all our APIs stateless.

To ensure that our system can scale as we intended we have in subsection 7.1.1 conducted throughput tests both with only a single node and dual nodes to test the effect of adding a node. From the results we can clearly see that our system can indeed be scaled out by increasing the amount of nodes. With 2 nodes the increase in performance seemed to correspond directly to the total amount of resources but without further testing it is difficult to say if this would hold as the number of nodes increases.

As the architecture is currently, the crawlers are part of the analysis component. Though the crawlers are handled in a way that should prevent errors in the crawlers from affecting the scheduler, the crawlers are still susceptible to failures in the analysis module. It might be beneficial to move the crawlers entirely into their own component, which would further insulate them.

8.2 Data handling

In this section, we discuss some of the limitations of our current data schema, storage strategy, and choice of DBMS.

8.2.1 Storage scalability

An important part of the scalability of our system is reducing the amount of data that we store. The basic entity relationship diagram for our stored data can be seen in Figure 5.7. As we see on the figure, a user can track a brand that is associated with a set of synonyms, all of which are mentioned in a set of SMPs.

However, while it is necessary for us to temporarily store SMPs in order to conduct KE over a time range, the statistics data (see section 5.5.1 for the contents and structure) only stores the top-k keywords and assigned sentiment for a synonym at a given time range. As such, we are significantly reducing the storage space required, allowing our system to scale better w.r.t. space.

However, we are still limited by the amount of data we can store in our current setup. Our chosen DBMS, PostgreSQL, does not support sharding out of the box, forcing us to scale vertically in case we need more storage space. This is not a maintainable strategy. Although PostgreSQL does support replication, which would allow us to distribute large request numbers to different nodes, a hard limit on storage space still exists.

As mentioned in subsection 5.5.2, the Citus extension to PostgreSQL effectively adds sharding to the DBMS. Our preliminary research into the extension shows that migrating from our current DBMS to a version that uses Citus will be a relatively easy process supported by underlying migration tools in Citus. Sharding would effectively remove the hard limit on storage space that we are currently set to incur.

Furthermore, sharding could significantly reduce query times when the amount of data stored grows large. Since queries are dependent on a time range, and based on some assumption regarding the usual time intervals that application users will query for, we could shard our data on the time column, e.g. by month. As such, when a user makes a query for a time interval, the nodes containing shards within that time span will be the only nodes doing work. Furthermore, this work can be done in parallel, further increasing the throughput.

In conclusion w.r.t. scalability, our current setup will incur a hard limit on storage space at some point in time if the amount of data continues to grow. This hard storage space limit can be mitigated completely by distributing the storage across multiple machines, i.e. sharding our data. Furthermore, sharding on specific attributes of our data, e.g. time, would allow us to increase our performance significantly as user queries could be processed in parallel.

8.2.2 Dynamic time ranges

In our current setup, we schedule keyword extraction every hour. Effectively, this means that the granularity of our statistics data is defined as keywords by sentiment by synonym by hour. A user of our system, such as a community manager, might only be interested in the exact times where the sentiment of their brand shifts, for which an hourly time interval may be too coarse to capture sudden violent changes.

Fortunately, as described in section 5.5.1, we have specifically designed our statistics data schema to be as dynamic as possible and support any time interval. Our statistics data objects are defined by a time range defined in terms of its beginning and end, and we impose no limitations on the granularity of these values. As such, given enough data and machines to distribute the workload across, our system will be able to schedule creation of statistics data on a minute-by-minute basis. Of course, the amount of work required will grow as KE will be calculated across all SMPs for all synonyms. As described in section 7.4, KE performance is affected primarily by the number of unique words rather than total number of words in a corpus and doing the KE on a smaller interval will thus not reduce the running time linearly according to the size of the interval. However, our system architecture supports the distribution of work to an arbitrary number of machines as mentioned in section 8.1.

8.2.3 NoSQL alternatives

As mentioned in subsection 5.5.2, we chose an SQL based DBMS for our system in part because of the superior query performance compared to NoSQL alternatives with highly relational data. While SQL DBMSs does not perform as well on insertion times, we expected that the number of queries would outweigh the number of inserts when we schedule KE on an hourly basis.

We just discussed the notion of dynamic time ranges and that our system both allows for dynamic time ranges in the data we store and allows our system to be scaled horizontally in order to provide the computing power required at an arbitrary granularity of time. However, if we were to schedule

KE on a minute-by-minute basis instead of hourly, the number of insertions would rise dramatically, and it is likely that our DBMS will become a bottleneck. In such a case, it is worth considering NoSQL solutions, such as MongoDB, as valid alternatives to our current DBMS. Since MongoDB is an inherently schema-less DBMS, we will not have to perform any checks for referential integrity when inserting data, increasing the performance during insertions. Although the query processing performance will obviously deteriorate in a NoSQL DBMS, much of the work can be distributed much in the same way that we plan to distribute our current DBMS with sharding, as discussed in subsection 8.2.1.

8.3 User interface

In section section 7.2 we described the usability test performed on the system. Only one test was performed which of course it is not enough to reliably determine whether our system is usable or not, but it did give us some insight into the usability of our system for a potential user. In this section we will discuss the results of the usability test and ways the system could be altered to improve the usability.

The first two issues are tightly coupled as they both revolve around the synonym creation process, where the test subject found it difficult to figure out where synonyms could be added to a brand. As seen in subsection 5.6.1, the current configuration requires the user to navigate to the page for a specific brand in order to add a synonym to the brand, whereas the test subject was trying to do it directly from the table of brands. Adding an option to the row options in the table of brands to spawn the synonyms component in a dialog, would potentially resolve the issue as well as reduce the amount navigation redirects the user would have to go through in order to add synonyms to multiple brands.

The third and last issue the test subject had was not being able to figure out that statistics for a synonym could be found by hovering over a point on the line for a synonym in the line chart. Once the test subject was shown how to do it, the subject found it very intuitive however, but the issue remains. One possible solution would be to add an extra statistics component. The statistics component would hold the statistics for each synonym at a point in time and have the user be able to change the date through an input field, or by clicking a date on the x axis of either graph, which would trigger the component to update. Another solution would be to have a guided walk through of the various components in the brand component on the first visit.

8.4 Sentiment analysis

In section 3.1 we found that SA can be used to provide useful information regarding public opinion. We presented several different SA approaches in section 5.7, where we chose to use a ML approach. From this we found some promising SA models, that we based our own implementation on, as described in section 6.3. These models are not necessarily the ones that achieve the highest accuracy, but models that performed well enough for our purposes, without being excessively complex. In section 5.7.4, we found that a simple CNN model can perform almost as well as state-of-the-art models.

All models were tested on their accuracy, loss and throughput in subsection 7.3.3. The NB model greatly outperformed the NN models in throughput, though the NN models still had high enough throughput to satisfy our demands, while achieving better accuracy and lower loss. We therefore chose to implement the *CNN-RNN* model as described in section 6.3. It could be possible for our NN models to perform even better in terms of accuracy and loss if we had spent time on hyperparameter tuning. We did not spend that much time on tuning as training all the different models was a very time consuming activity that did not significantly increase our quality of service.

We expected that CNN and RNN models would extract different features which our tests seem to indicate as described in subsection 7.3.4. Furthermore we also expected our NN models to perform better than NB, as they should be able to extract some dependencies between the words in a document. Only the two CNN models using striding achieved worse results than the NB, indicating an inability to learn inter-sentence dependencies between words that can be used effectively for determining sentiment.

We expect that our model will perform similarly to the results seen in subsection 7.3.2 when in production. Since our model has an F1-measure that is almost equal to its accuracy, we can say that its recall for the positive set is equal to its recall on the negative set, described in subsection 7.3.4. The keyword extraction algorithm would therefore get an equal amount of correctly classified positive and negative documents. As we achieved 82% accuracy and F-measure, a big part of the documents are correctly classified, meaning the keyword extraction algorithm would be able to find the correct keywords.

8.5 Keyword extraction

We present a set of different KE approaches in the design chapter (see subsection 5.8.1 and subsection 5.8.2). The graph based approach, which is based on TKG introduced by [2], is further extended with a set of different centrality measures and weighting strategies that we expect to increase performance.

The approaches are implemented in section 6.5, and are evaluated in terms of their running time and precision in section 7.4.

Part of the solution to the problem definition stated in section 3.4 requires that we are able to extract keywords from a large corpus of SMPs. The corpus size is expected to grow very large, and our chosen approach must be able to handle large data volumes. As seen in subsection 7.4.1 where we compare the running times of the TFIDF and KG approaches, we see that, while the TFIDF approach and KG using PageRank approach do not increase dramatically in running time as the amount of data grows, the KG approach using closeness centrality, as used in [2], completely fails to terminate within 5 minutes when the number of documents grew beyond ~ 200. A short analysis of the time complexity of the closeness centrality calculations show that the approach is not fit for this system, as the worst-case running time of the algorithm grows polynomially ($O(N^3)$) where N is the number of unique words in the corpus.

It may have been worth to test the precision of the KG approach using closeness as well as the other approaches despite its running time. However, an important aspect of our system is that we are able to provide users of our system with data regarding brands collected in a timely manner. We have observed snapshots where the number of documents per brand far exceed ~ 200, and we cannot have the KE process taking several minutes for each brand if we want keywords for every brand for every hour, as is our current setting.

However, it is important to note how the number of unique words grows as the size of the corpus grows. A short experiment conducted in subsection 7.4.1 showed that the number of unique words converge (in the data set we used) at ~ 9000 words encountered first at ~ 300 documents. As such, as the number of documents grow beyond 300 the number of unique words would not increase dramatically. This finding allows us to further consider using closeness centrality as the approach to use, given that it extracts keywords with a higher precision than the other approaches. Because we can expect the running time of the closeness centrality calculation to stay relatively unchanging beyond 300 documents, we can look into distributing the complex computations across several machines, for example using the MapReduce programming model. For example, we might split up a large corpus at feed the splits to several machines all running KE, and then combine the results according to the frequency of the keywords extracted. Other more sophisticated approaches to distributing the calculations may also be possible. However, it is important to note that extracting keywords based on their collective frequencies over a set of documents may not work as well as extracting them from one single graph, as we have seen with TFIDF.

Another interesting observation in the performance comparison experiments was that our added weighting strategies (using absolute frequency difference and inverse absolute frequency difference between two nodes as the weight of their connecting edge) did not increase the precision of KE using PageRank. PageRank, in its identically weighted implementation, models the transition probability from one word to another connected word as a uniform probability distribution between all the out-edges. In the weighted version, which is what we use when weighting edges according to frequency differences, we adjust the probability distributions according to node similarity. As such, if we use absolute frequency difference, the transition probability of going from one node to another would decrease if the words were very similar. Inversely, using the inverse frequency difference, the transition probability would increase if the words were very similar.

Since none of these weighting strategies afforded higher precision scores than the identically weighted version of PageRank, we can conclude that frequency difference between words, whether absolute or inverse, is not a feature that is suitable to be used with PageRank to extract keywords effectively. However, we believe that adjusting the transition probability between words according to some notion of word similarity could be beneficial in combination with using PageRank for KE, and we would like to explore this in depth in the future.

8.6 Development process

In this section, we discuss the various steps we have taken towards structuring, planning, and improving the quality of our development process, how they have worked for us, and how we can further improve in future projects.

8.6.1 Scrum

At the start of this project, we decided to apply some Scrum practices that we have had good experience with in previous projects. In this section, we discuss the degree to which we have implemented them correctly, the benefits, and how our approach to agile development can be improved in the future.

Daily meetings

Throughout the entirety of this project, we have conducted daily stand-up meetings at the beginning of every day that was allocated completely towards project work. During the meetings, all group members stated what they worked on the day before, what they were going to be working on today, and their roadblocks, if any.

To ensure that we wouldn't forget the meetings, we have used both a daily Slack notification as well as a audible alarm. Especially the audible alarm has proved effective at ensuring we held the meetings on time.

The meetings have proved to be effective, as all group members would be aware of the current work being done and the currently relevant complications from the beginning of the work day, allowing everyone to re-think their approaches to their own tasks in order to accommodate for complications or roadblocks if necessary. Furthermore, it served as an effective tool for knowledge sharing between group members, as task-specific details were covered and explained during the stand-up meeting.

One issue with our daily meetings has been that not all group members were well prepared to state their current work and status when the meeting began, causing the meeting to take longer while the rest of the group enquired about the details.

We will be using stand-up meetings in future projects, as they have served as a good tool for managing tasks and project status on a daily basis. We will also aim to implement a way of ensuring that all group members are prepared to state their work and status clearly when the meeting begins.

Kanban board

As covered in the development process sections (see chapter 2), we decided to implement the use of the GitHub Kanban board to keep track of tasks and task statuses instead of using a Scrum board.

While the Kanban board was used during the beginning of the project, group members gradually stopped using it as the project matured. There are several reasons for this, and we will discuss those reasons and their possible solutions here.

First, the Github Kanban board was created for all projects. As such, the board quickly grew very large, as we have organised our system into several sub-repositories for which all tasks were combined into the same Kanban board. This made the board too crowded in order to easily derive the status of the project and, more importantly, the status of the sub-repositories.

Secondly, there is no built-in functionality for assigning a specific task to a specific group member in the GitHub Kanban board. This required that group members were to write their name in the description of the task to prevent two people working on the same thing. Another issue regarding assignment of tasks was that there was no built-in functionality for seeing which group members had completed the task and moved it to review, who had reviewed it, and so on. Because of this, every time a task was incorrectly implemented, or a task was not clearly defined, group members had to interrupt all other group members in order to find someone who knew more about it.

In the future, we will try to find a project management tool supports the functionality of assigning a task to a specific group member, and allows group members to track the work that had been done throughout the lifespan of the task. We will also construct boards for each separate project rather than a single catch-all board for the entire system.

8.6.2 Test-driven development

We have employed TDD to good effect in previous projects, and decided to do it again in this project in section 2.2. Unfortunately, only some of the sub-repositories in the project were implemented according to the TDD principles which required us to write tests for the other sub-systems after they were implemented. This is of course not optimal, as the sub-systems that were developed in accordance with TDD were much easier to diagnose in case of run-time issues. The reason behind tests not being written for sub-systems before implementation was a combination of not being sure of the expected result, not prioritising tests because of the inherent simplicity of the functions to be implemented, and simply forgetting. In order to make sure that everyone knows what the expected result of any functionality in the system should be, we should put more effort into designing the system and presenting said design in a comprehensible way to all group members. We acknowledge that some experimental implementation can be useful in order to understand what the result should be in terms of types and values, but tests should always be written before the actual functionality is implemented.

With regards to group members simply forgetting to write tests, we expect an effective way to combat this is to employ pair-programming. This will ensure that more appropriate tests are written, the work can be done in parallel, and it will reduce the likelihood of a group member forgetting to write tests completely.

Unfortunately, we have not conducted Continuous Integration (CI) for this project, which is an effective way to expand upon the usefulness of TDD. We only ran the test suites whenever our system was facing problems, for example when the Trustpilot crawler started failing (as is seen in Table 7.7). If we had instead only allowed group members to push their code to the system when all tests succeeded, we would have been made aware of the such issues much sooner, and would use our regression test suites to full effect.

As a final note, we did not define a common standard for writing test, including the directory structure, how we distinguish between unit- and integration tests, and more. The lack of a common testing guidelines was exacerbated by our failure to do code reviews of our unit tests as we intended to, as stated in section 2.2.

In the future, we will put more effort into ensuring that tests are written before implementing functionality. We will also put more effort into designing the functionality and structure of the system and presenting said design in a comprehensible way so all group members intuitively understand how the system works. Finally, we will employ pair-programming on all major tasks in order to improve the quality and quantity of the tests written.

8.6.3 Development server

By our request, the university issued a server to us that we have used as a development server throughout the project. The purpose of the server has been to test how our system would work in a simulated production environment.

The server has proven invaluable to the development of our system. It has provided the project with a common basis of functionality, doing away with different systems behaviours caused by different Operating System (OS) and/or hardware found on each group member's computer.

Furthermore, the server allowed all group members to use the system simultaneously as if in production, conduct throughput tests, and to test the effect of adding computational nodes to the system (i.e. splitting requests between a group member's computer and the server).

One major drawback of the development server is that a large amount of work and maintenance is required in order to keep the server running. Although several hours have gone into server maintenance, we regard the time lost as well spent for the developmental confidence the server has provided the group with during this project.

If a server is applicable, we will also be using a development server for our next project.

8.7 Fulfilment of system requirements

This section discusses the degree to which we have fulfilled the system requirements proposed in chapter 4. The discussion argues mainly from thoughts made during the design phase and from results gathered in chapter 7.

Analytical module requirements

In the following, we enumerate all functional system requirements regarding our analytical modules and argue whether or not the requirement has been fulfilled. The requirements are stated in section 4.1.

1. **The system must be able to determine the sentiment and keywords- and phrases from a set of SMPs regarding a specific brand or topic:** In section 5.7 and section 5.8, we analyse methods of determining the sentiment and keywords of a text. We decide on a set of different approaches, provide our own extensions to the approaches we base them on, and baselines and implement them in chapter 6. The approaches are evaluated, and the best models for our system are chosen based on several metrics of comparison in section 7.4 and subsection 7.3.4. Ultimately, our sentiment analysis model is able to determine the sentiment of a post with 82% accuracy, and our keyword extraction module is capable of extracting keywords from a corpus with 80% precision, and with 61% accuracy when extracting keywords from single texts. On a scheduled basis, the system calculates the sentiment of all gathered textual data, after which keywords are extracted from the posts. Given that the resulting pairs of sentiments

and keywords are related to a synonym and are grouped according to the brand they represent when a user queries the statistics for a brand, we regard this requirement as fulfilled.

2. **The system must be able to group and summarise keywords based on SMPs assigned sentiment:** As described in subsection 6.1.2 which covers the implementation of the scheduler, the scheduler groups posts according to our sentiment classes before performing KE on them and aggregating them in snapshots. This information is later available when querying the statistics service and thus we regard this requirement as fulfilled.
3. **The system must be able to scale and handle large amounts of data and queries without becoming unusable:** In subsection 7.3.4, we choose the best sentiment analysis model to use in this system based on accuracy, precision, recall, F-measure, and throughput. The chosen model, while not affording us the largest throughput, afforded predictions with 82.0% accuracy (full results can be seen in Table 7.3). In subsection 7.4.1, we compare the KE approaches in terms of their running time. We exclude using closeness centrality as the centrality measure due to the algorithm taking too long with a large number of documents. The remaining approaches are evaluated in terms of precision, and the graph-based approach used identical weighting and PageRank as the centrality measure is chosen as the best approach for this system. Both of these choices regarding our analytical modules directly serve to make our application as load tolerant as possible. The system architecture (see Figure 5.1) is structured according to a service-oriented architecture. All components of the architecture are implement as self-contained and re-deployable services, allowing the system to scale horizontally by adding more instances of each component in order to distribute heavy computations. We use a PostgreSQL DBMS that does not have in-built support for sharding, but describe an extension to the DBMS, Citus, that adds sharding to the DBMS, allowing us to scale our storage horizontally while also increasing performance w.r.t. response times and throughput. Finally, we evaluate the throughput of our system in subsection 7.1.1, showing that the response time of our system does not deter for larger numbers of concurrent users. We also show that, by adding more nodes to the system architecture, we can handle even more concurrent users with consistent response times. Given the focus on scalability and load tolerance throughout all parts of this project and the results shown in chapter 7, we regard this requirement as fulfilled.
4. **The system should provide an easy-to-use API allowing separate client systems to use it, including but not limited to our own web application:** In hindsight, being easy-to-use is quite a vague requirement, however as described in subsection 5.4.10 we have required all APIs to use the API documentation tool Swagger. The tool generates an online documentation wherein developers are able to see expected input and possible response codes, which we believe eases development. In principle, any system capable of performing HTTP requests is able to use our APIs and thus the requirement is regarded as fulfilled.
5. **The system should dynamically gather and analyse opinion-rich SMPs regarding all relevant brands and topics in the system:** The system flow described in subsection 5.3.6 describes in detail the design of the scheduler that allow the system to dynamically gather and analyse SMPs regarding all relevant brands and topics in the system. The scheduler service implementation described in subsection 6.1.2 goes on to describe the implementation of the scheduler that we have used to fulfil the requirement.

8.7.1 Web application requirements

In the following, we enumerate all functional requirements for the web application part of the system built in this project, and determine whether or not the requirements have been fulfilled. The requirements are stated in section 4.1.

1. **The system must be able to comprehensibly visualise analysis results to a user:** In section 5.6, we design the UI to be presented to users of our system. The UI is implemented in section 6.2, and the results of a brief usability test in covered in section 7.2. The usability test uncovered some potential flaws in the user experience design of our UI specifically regarding the intuitiveness of the interaction. The usability test was only conducted with a single person, so we cannot regard these results as general usability issues. We have not developed the system in this project for a specific end-user or demographic, and our specific requirements for end-user functionalities were gathered from a set of synthesised user stories (see section 5.1). We believe that the representation of how sentiments and keywords change over time through the use of line-charts and histograms provide an intuitive overview over a complex set of data. The intuitiveness of the data representation is also backed by the results of our usability test. As such, we regard this requirement as fulfilled, though we acknowledge that ideally we would do significantly more testing before a potential release.
2. **The system must be designed in accordance with commonly approved user-interface design patterns:** The UI as demonstrated in section 6.2 is designed in accordance with Google's Material Design language[1]. All of Google's UIs adheres to the design patterns proposed by the Material Design language and thus by adhering to the same design patterns we regard the requirement to be fulfilled.
3. **The system must store historical analytical data such that the user can view changes in sentiment and topics over time:** In section 5.5.1, we present the structure of our statistics data which contains keywords extracted from a corpus of opinion-rich SMPs sorted by the assigned sentiment of the posts they were extracted from. In Figure 6.4, we present an example o a user query response shown as a line chart where the line represents the change of sentiment towards the user-defined brand over a specified time range. Furthermore, the user has the ability to hover over the sentiment at any point in the graph in order to see the keywords extracted at that point in time. As such, the user is able to see both developments in sentiment and topics for a user-specified brand over time. As such, we regard this requirement as fulfilled.
4. **The system must allow users to track multiple brands or topics with associated search keywords:** When designing the entities in the system, this requirement was accounted for by associating each account with a number of brands (see Figure 5.7). A brand is then associated with a number of synonyms (i.e. search keywords), which may implicitly be shared with the brand of other accounts. As such, we regard the requirement as fulfilled.
5. **The system should allow users to compare the development in sentiment for different brands:** Currently, it is not possible for an end-user to directly compare the sentiments of two brands. While the API responsible for serving this information to the user is implemented and working, the UI does not present an end-user with this functionality. The only way an end-user can compare the sentiment of two brands is by querying both brands in separate windows and visually comparing the graphs. As such, we consider this requirement as partly fulfilled.
6. **The system must allow users to specify different search keywords for the brands they**

[1] https://material.io/

are tracking: As described in section 5.5, a brand in our system is represented internally as a set of synonyms for the brand, i.e. different search keywords for the brand. The synonyms are used as the basis for information gathering in the web-crawlers, and all SMPs regarding a synonym are also considered as related to the brand(s) that include the synonym. When a user queries the statistics for a brand in our UI, they are presented with a visualisation of the aggregated results of all synonyms that the user has entered for the brand, as seen in Figure 6.4. As such, we regard this requirement as fulfilled.

8.7.2 Non-functional requirements

In the following, we enumerate the non-functional requirements for our system stated in section 4.2 and determine whether or not these requirements have been fulfilled.

1. **Scalability:** As mentioned for the previous requirements, our system architecture is structured according as a service-oriented architecture where all architectural components are deployed as self-contained services that expose a number of APIs that are used by the rest of the system. As argued in section 8.1, this structure allows us to add more instances of each component in order to increase the throughput and computational power. Furthermore, we show in subsection 7.1.1 that our system is indeed able to distribute the computational workload across multiple machines, increasing the throughput and number of concurrent users that the system can handle. As such, our system directly supports horizontal scaling w.r.t. scaling for computational power. We chose PostgreSQL as our DBMS in subsection 5.5.2 which does not have built-in support for sharding, which is a typical method of scaling storage space horizontally. While PostgreSQL does not support this, we consider an extension, Citus, to which we can easily migrate our PostgreSQL DBMS in order to allow sharding across multiple nodes. As discussed in section 8.2, such sharding would not only mitigate the inevitable storage space limit that we currently set to incur, but could also be used to improve performance w.r.t. response times. Finally, as argued in the fulfilment of the functional requirements, the approaches for conducting SA and KE have both been chosen from a combined consideration of precision, accuracy, and scalability as the amount of data grows. Given that every part of our system has been designed with scalability as a constant factor of consideration, and further given our results in the performance of our analytical modules and the stress test of our system (see subsection 7.4.1, subsection 7.3.3, subsection 7.1.1), we regard the requirement of the system being scalable as fulfilled.
2. **Robustness:** Since the components of our system are built as self-contained services, the rest of the system will remain unaffected in case one service fails. For example, if the SA service fails in production, users will still be able to make queries, and the information gathering services will keep scraping sentiment-rich SMPs from our select social media sites. Through the use of Docker, services are easily re-deployed in case they fail at run-time. As such, we regard the requirement of the system being durable as fulfilled.
3. **Performance:** As mentioned in section 4.2, we evaluate the performance of our system (e.g. its sluggishness) in terms of its load tolerance. In interest of not repeating ourselves, every part of the system has been built with scalability and load tolerance in mind, and the results of our efforts in this regard afford a median response time of $\sim$ 100 milliseconds even at several hundred concurrent users on a single machine. By distributing the workload across multiple machines, we are able to handle even more concurrent users at the same response time

(see Figure 7.1, Figure 7.2). As such, we regard the requirement of performance regarding responsiveness and load tolerance as fulfilled.

4. **Usability:** As discussed in subsection 8.7.1, we can only consider this requirement as partly fulfilled as we do not have any actual end-users or demographic to test this system against. While a brief usability test has been conducted for the system and showed promising results despite a few potential flaws regarding the intuitiveness of the UI, a single participant is not enough in order to consider the results of the test as general usability issues. Generally, the design was intuitive for the participant, and since this system has not been designed for any specific end-user, we consider this requirement as fulfilled, though we acknowledge that further usability testing is required before a release of this system can be considered.
5. **Testability:** Following the discussion in subsection 8.6.3, we have not followed TDD to the degree we initially planned to do. As such, several of our system components remain either untested or at very low code coverage. Indeed, some issues in our system (such as the Trustpilot crawler failing due to website restructuring) could have been caught much earlier had we tested our system appropriately as discussed in section 7.5 and subsection 8.6.3. As such, we do not consider this requirement as fulfilled.
6. **Privacy:** As described in section 5.5, we do not store any user information in in the statistics data we show when a user queries a brands. The only point at which we store user information (such as user names or IDs) is in our information gathering services were we use this information to avoid storing duplicate posts. In this case, the user names are hashed in order to increase the privacy of the users from which we gather data. As such, we consider this requirement as fulfilled.

In section 5.8 and section 5.7, we conduct an analysis of existing approaches to both SA and KE. In subsection 5.7.3 and section 7.4.2, the most appropriate approach is chosen and subsequently designed. In section 6.3 and section 6.5, we cover the implementation of these functionalities. As shown by our test results in chapter 7, our system is successfully able to calculate the sentiment over a set of posts distributed throughout a time range. Furthermore, sentiments are calculated individually for posts specifically regarding a specific topic or brand, which is referred to as a `synonym` in our design and implementation. While the SA performance has been evaluated directly from its accuracy over our test set during training (see section 7.3), we have not evaluated our KE functionality. While it could be possible to test the extracted keywords against training data sets for machine learning KE models, the optimal strategy to test our KE would be to test it against keywords generated by humans. There are multiple difficulties regarding this, with the first one being that persuading enough people to read a piece of text and state their keywords for it is a financially expensive affair. However, the main objective of this project has been to create a system that is capable of conducting computationally complex operations (e.g. SA and KE) in a scalable manner, and not to improve over existing solutions for said computations. While we acknowledge that conducting an actual evaluation of the KE, we regard the requirement of being able to extract keywords fulfilled, since we have implemented the approach for KE that best fit our problem as decided in section 7.4.2.

Chapter 9: Conclusion

In this chapter, we summarise the discussion in chapter 8 by evaluating our system as a potential solution to the problem definition defined in section 3.4.

We have developed a system that should provide a solution to the problem defined in section 3.4. Our system is capable of continuously gathering data from social media platforms by means of continuous web crawlers whose search relies on user-specified brands. The brands specified by users are internally modelled as a collection of synonyms which allows users to define brands at any level of sub-topic or sub-brand granularity. On a scheduled basis, the system calculates the sentiment of all gathered posts and, at set intervals, extracts keywords for all posts related to synonyms with the same assigned sentiment. The resulting statistics data can be queried by the user through use of a web application, allowing users to visualise the change in sentiment over time for a brand as well as see the keywords extracted from the posts at any point in time within the time span the brand has been recorded.

We have tested our SA and KE modules. The SA model is able to classify documents with a 82% accuracy and classifies documents fast enough for our purposes. As the majority of our documents are classified correctly it is possible for the KE algorithm to extract the relevant keywords. The KE algorithm has a precision of 80% and is faster than the other proposed models. We are therefore able to group and summarise keywords for SMPs with assigned sentiment in a timely manner based, allowed system users to see the change in sentiment and topics for a brand over time.

As our solution is developed with scalability in mind, a major focus was to make our services as decoupled as possible. With the exception of the scheduler being tightly coupled to the crawlers, our services achieve this goal by implementing an API layer for each service. These services can then be deployed on multiple nodes in a cluster such that load balancing between services enables the opportunity of scaling out the system, which in some cases can be preferred over scaling up a node. With our choice of using PostgreSQL as our DBMS, we currently have a limitation in how much we can scale out. Natively, PostgreSQL only supports data replication, but given enough data we could eventually reach a limit in storage capacity. We have proposed the PostgreSQL extension Citus as a solution to this bottleneck, which should be relatively easy to migrate to.

Given that we have fulfilled almost all system requirements proposed in chapter 4 with the exception of the non-functional requirement of testability, we consider our system, SentiCloud, as a complete solution to the problem stated in section 3.4.

Chapter A: Preliminaries

Reading the preliminaries will make it easier to read the report as it gives background for technologies used and topics covered later in the report. The goal of the preliminary is to give the reader a deeper understanding of these technologies and topics as well as the theory behind it before delving deeper into the actual design and implementation of said technologies and theories. We will describe three different ML models in section A.1, section A.2, and section A.3, as these will be used in subsection 5.7.4. Furthermore, we describe the underlying principles some of the KE approaches in section A.4. Lastly, we describe some different accuracy measures in section A.5, that will be used to evaluate the different algorithms and models.

A.1 Convolutional neural network

This section will describe CNN, their impact on image recognition, why they achieve great results within this area, how this can also be applied to text classification and finally how they work.

CNNs are widely used for image recognition and the model structure has achieved great results within this area. The use of CNNs for image recognition was popularised by [27] as they achieved a 10 percent lower top-5 error rate than number two contender in the Large Scale Visual Recognition Challenge (LSVRC) using the CNN model in 2012[1]. In 2017, the majority of submissions to the LSVRC used CNNs in their model.

The reason CNNs excel in image recognition is because they can handle un-centered images, as it can use the spatial topology of nearby pixels to extract features [28]. Pictures often have many features, both in number of pixels and colours. A Fully Connected Neural Network (FC) would therefore have many tens of thousands weights with only a hundred hidden units. Furthermore, they are not capable of capturing the invariance of local distortion [28]. The input should therefore be preprocessed shuch that variations in the input data is ignored. With a sufficient sized network and enough training data, it could be possible for a FC, though time and hardware might be significant restrictions. As CNNs have weight replication, they automatically obtain shift invariance, meaning it can handle some displacements and variations of the input. Images have a topology, as pixels spatially near each other are correlated. A FC ignores the order of the input as long as the order is fixed. A CNN extracts these local features having the receptive fields of hidden units locally.

Sentences are both spatially and temporally ordered. It therefore makes sense to use CNNs to extract

[1] [28] also had a great influence on CNNs and their popularisation

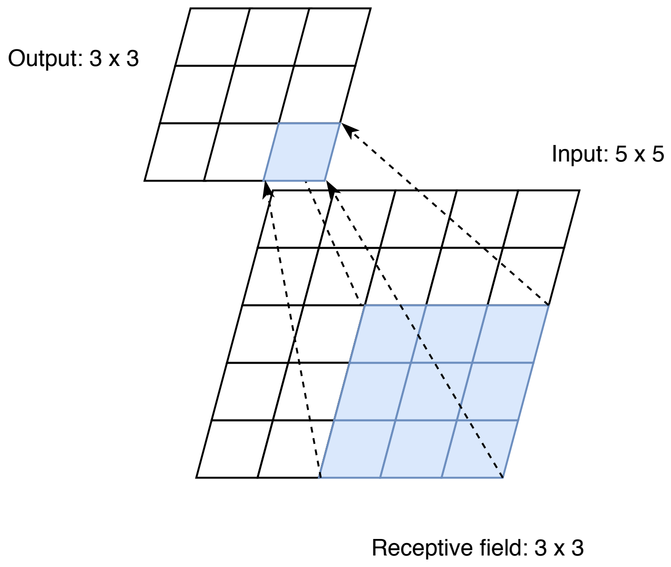


Figure A.1: A visualisation of the receptive field for a feature.

local spatial features. The RNN (see section A.2 for detailed description) would be better to extract long time dependencies, that the CNN would not extract [52]. Similarly, the RNN could ignore strong sentiment in a key phrase that the CNN can detect. Even a simple CNN model has already achieved good results for sentiment classification [26].

The CNN uses a receptive field on the input to calculate an output feature [28]. A receptive field is an area off the input that is connected to the output unit. The thought is that the same high level constructs that appear in one location are likely to appear again in another location. If the units with different receptive field therefore share weights they will all be able to extract the same feature for all fields [28]. The units sharing weights make a layer, as in Figure A.1, and are referred to as a feature map.

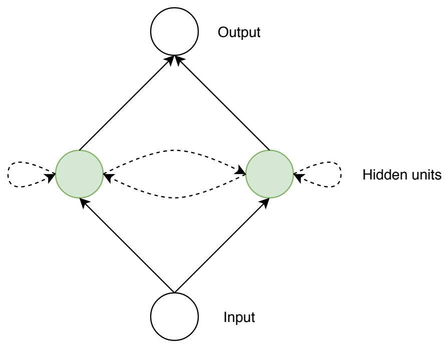
There are often multiple feature maps, such that multiple features are extracted from each location. The value calculated for a unit c, given a receptive field x and a weight matrix w, is

$$c = f(w \cdot x + b),$$

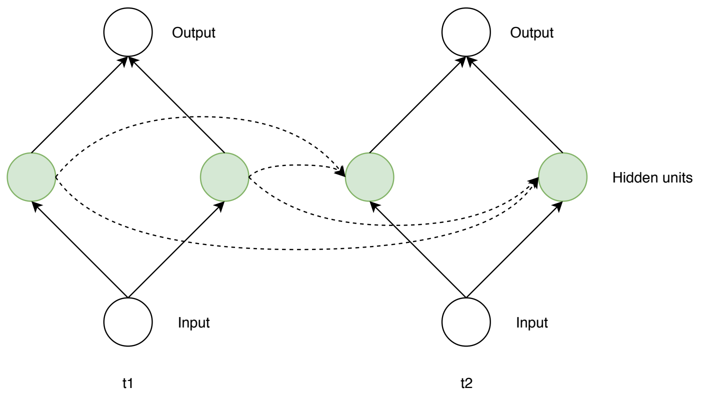
where f is a non linear function and b is a bias [26].

Figure A.1 illustrates a CNN where the receptive field is a 3×3 matrix and a stride of 1×1. The stride descripes how much the receptive field changes for each unit compared to the adjacent unit.The unit to the left of the marked unit would therefore have a receptive field projected one column to the left. A stride of 2×2 would then mean that every second output unit is ignored.

Suppose that the input is shifted a bit, then the output would be shifted an equal amount [28]. The exact position of a feature is less important after extraction, and could be harmful to keep if there is some distortion and shifts of the input. A solution to this problem is subsampling. Subsampling performs local averaging of a feature map thereby reducing sensitivity to distortions and shifts.



(a) A figure of a standard RNN model.



(b) A figure of an unfolded RNN model.

Figure A.2: The figures illustrate the RNN model architecture, where a dashed line represents a recurrent edge and a solid line represent for each timestep an activation in a feedforward neural network.

A.2 Recurrent neural networks

This section will describe RNNs, what their strength is and how they work. Furthermore, we will describe to extension of the traditional RNN, as RNN has a problem with exploding and vanishing gradients.

Speech and texts are expressed as temporal sequences [17]. Extracting the semantic dependencies over a time period is therefore necessary to correctly classify and analyse texts. There have been attempts to solve this problem by representing time spatially without promising results [17]. RNNs try to solve this problem by representing time implicitly.

The principle behind RNNs is that the output of adjacent timesteps is used as input to the next timestep to pass important information through the network [30]. The edges from adjacent timesteps are called recurrent edges. Recurrent edges can form loops, as shown in figur Figure A.2a. At a given timestep t, the nodes with recurrent edges would receive input $\boldsymbol{x}^{(t)}$ and from $\boldsymbol{h}^{(t-1)}$, where, given some integer k, $\boldsymbol{x}^{(t)} \in \mathbb{R}^k$ is a real valued vector, and $\boldsymbol{h}^{(t-1)}$ the output vector of the hidden nodes from the previous timestep. The value of a hidden node is then calculated as

$$\boldsymbol{h}^{(t)} = f(W^{hx}\boldsymbol{x}^{(t)} + W^{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h),$$

where $h_j^{(t)} \in \boldsymbol{h}^{(t)}$ and W^{hx} is a normal weight matrix from input to a hidden node, W^{hh} is a recurrent weight matrix from the time adjacent hidden nodes to the current. All weights defined like this are shared across time for each hidden node. f is some activation function, such as sigmoid or rectified linear unit. At $t = 0$ $\boldsymbol{h}^{(t)}$ is set to some standard weight [12]. The output at some timestep is calculated as

$$\hat{\boldsymbol{y}}^{(t)} = f(W^{yh}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y).$$

The RNN can be unfolded as seen in Figure A.2b, making a layer for each timestep. It is therefore possible to use backpropagation, called Backpropagation Through Time (BPTT).

A problem with RNNs is the vanishing and exploding gradients, which occurs when backpropagating over many timesteps, meaning that the RNN either ignores the first word or it is the sole determining factor. This is because some input at time t to some error calculated at d, the contribution of the input at time t to the output of d might explode or vanish exponentially fast as the distance in timesteps between t and d increases [30]. The problem can be solved with extensions of the RNN architecture that introduce mechanisms to decide what information to pass on between timesteps.

A.2.1 Long Short Term Memory

LSTM is a solution to the vanishing gradient problem [30]. The intuition behind LSTM is that the state of the previous hidden nodes state is connected with a recurrent edge directly to the current, with a constant weight, ensuring no vanishing or exploding gradients.

Each node in the hidden layer of the LSTM is replaced with a memory cell. A memory cell is represented as one of the square boxes showed in Figure A.3 and can be described as a collection of nodes connected in a specific way. They would be connected as they are in Figure A.2b for multiple nodes in the hidden layer. The memory cell has gate nodes, which can limit the flow of information by multiplying the gate together with another node. The idea is if a value is multiplied by zero, no information is passed through. This is used in Equation A.2 described later.

The memory cell consist of five nodes. The input node, labelled with g in Figure A.3, takes the input of the current input and has a recurrent edge to the previous output of the hidden units. The value of g is calculated as

$$\boldsymbol{g}^{(t)} = \phi(W^{gx}\boldsymbol{x}^{(t)} + W^{gh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_g^{(t)}),$$

where a bold letter with the superscript (t) represents the output of all nodes of a single type at timestep t, as a vector. The activation function in the orignal paper was sigmoid, though most modern implementations use tanh as it seems to converge faster in practice [30]. The next is the input gate, labelled i, and as the name suggests, is a gate for the input node. Its purpose is to learn when to let the internal state (described momentarily) get information from the input. The input gate formally defined as

$$\boldsymbol{i}^{(t)} = \sigma(W^{ix}\boldsymbol{x}^{(t)} + W^{ih}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i^{(t)}).$$

The forget gate makes it possible for the memory cell to flush its internal state, meaning its memory. The forget gate, labelled f in the figure, is defined as

$$\boldsymbol{f}^{(t)} = \sigma(W^{fx}\boldsymbol{x}^{(t)} + W^{fh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f^{(t)}), \tag{A.1}$$

The internal state is making it possible for error to flow across timesteps without exploding or vanishing. There was no forget state in the original paper [22], so the internal state had a recurrent edge to itself. The internal state, labelled s, is defined as (with a forget state)

$$\boldsymbol{s}^{(t)} = \boldsymbol{g}^{(t)} \odot \boldsymbol{i}^{(t)} + \boldsymbol{f}^{(t)} \odot \boldsymbol{s}^{(t-1)} + \boldsymbol{b}_s^{(t)}). \tag{A.2}$$

This leads to the output gate (o), which learns when to let out values from the internal state. It is defined as

$$\boldsymbol{o}^{(t)} = \sigma(W^{ox}\boldsymbol{x}^{(t)} + W^{oh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o^{(t)}), \tag{A.3}$$

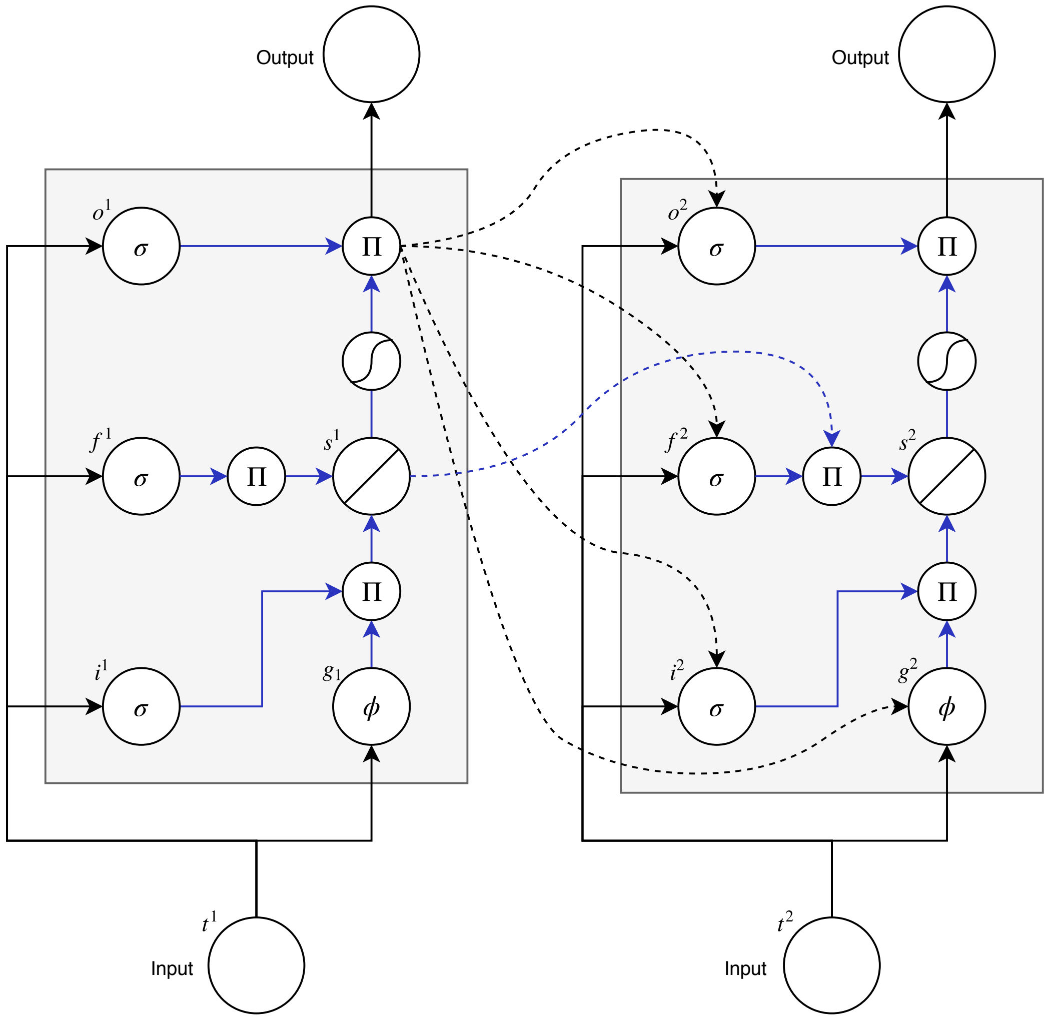


Figure A.3: A visualisation of the interaction between memory cell at two adjacent timesteps. Blue arrows represent connections with a fixed weight of 1, black arrows have weights and dotted lines are recurrent edges. The superscript represent the timestep and Π is pointwise multiplication.

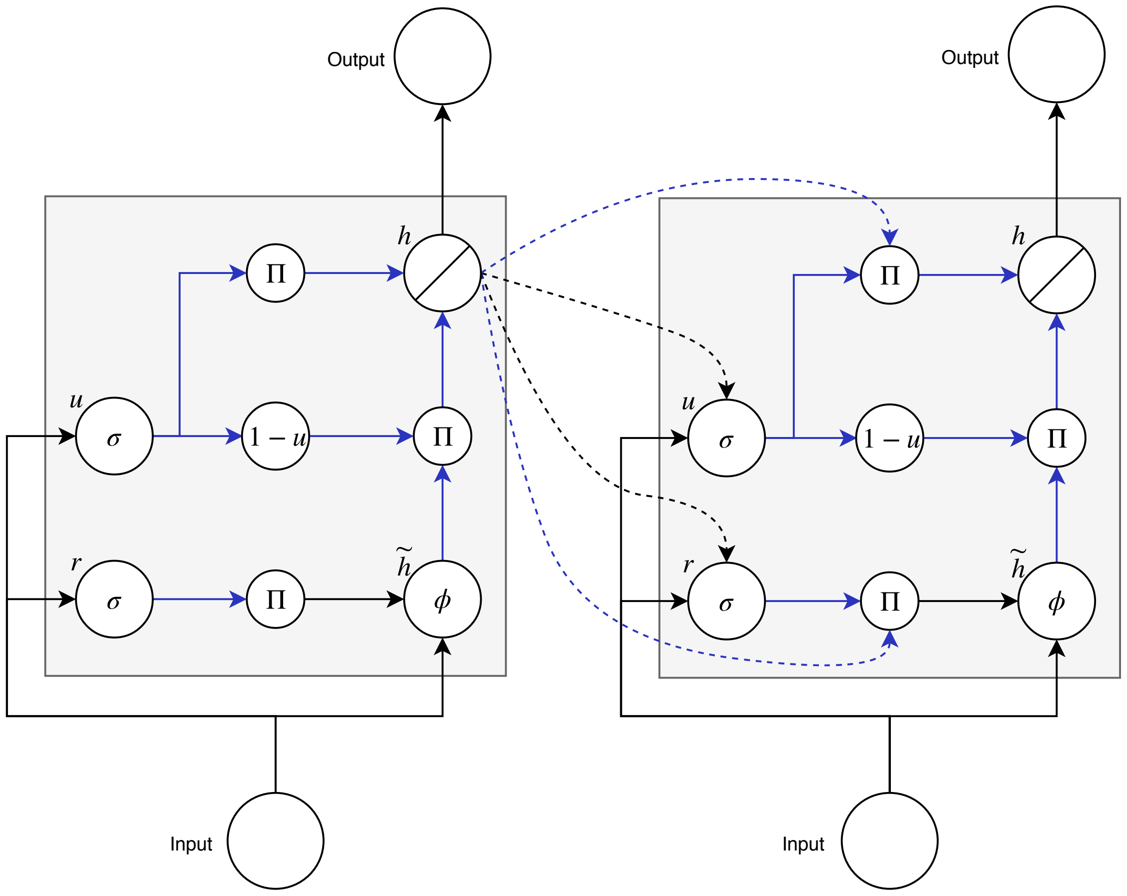


Figure A.4: A visualisation of the interaction between gated recurrent units at two adjacent timesteps. Blue arrows represent connections with a fixed weight of 1, black arrows have weights, and dotted lines are recurrent edges. The superscript represent the timestep and Π is pointwise multiplication.

which is multiplied with the internal state, to produce the output of the memory cell. The output of all memory cells or hidden nodes is then

$$\boldsymbol{h}^{(t)} = \phi(\boldsymbol{s}^{(t)}) \odot \boldsymbol{o}^{(t)}.$$

The gates of a memory cell therefore learns when to let error in and out. The problem of the vanishing gradient is solved as the error does not decay when the gradient is sent back.

A.2.2 Gated recurrent unit

This section describes the GRU, which tries to solve the vanishing gradient problem like the LSTM. As can be seen in Figure A.4 the GRU arcitecture resembles that of the the LSTM (Figure A.3).

GRU has two gates, one is the update gate, the other is a reset gate. The update gates function is controlling how much the unit should update its activation, where the reset gates purpose is to allow the unit to forget previous calculations such that it can generate a more compact representation [12], [10].

The reset gate is calculated as

$$\boldsymbol{r}^{(t)} = \sigma(W^{rx}\boldsymbol{x}^{(t)} + W^{rh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_r).$$

The update gate allows, like the LSTM forget gate (Equation A.1), the unit to decide what information should be carried over to the next timestep. The update gate u, referred to as z in the original paper [10], is calculated as

$$\boldsymbol{u}^{(t)} = \sigma(W^{ux}\boldsymbol{x}^{(t)} + W^{uh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_u).$$

A unit exposes the whole content of its memory, as shown in the equation below, while the LSTM forget. The GRU adds the previous values to the new like LSTM, making it possible to remember features over longer periods of time. If a feature is deemed important by the forget gate in the LSTM or the update gate in GRU, then that an important feature is not overwritten [12].

$$\boldsymbol{h}^{(t)} = \boldsymbol{u}^{(t)} \odot \boldsymbol{h}^{(t-1)} + (1 - \boldsymbol{u}^{(t)}) \odot \widetilde{\boldsymbol{h}}^{(t)}.$$

The hidden state $\widetilde{h}$ uses the reset gate, capture short term dependencies in one hidden unit and longterm in another [9]. The hidden state is calculated as

$$\widetilde{\boldsymbol{h}}^{(t)} = \phi(W^{hx}\boldsymbol{x}^{(t)} + W^{hm}(\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)}) + \boldsymbol{b}_h),$$

where m in w^{hm} is the edge from the pointwise multiplication of $\widetilde{h}^{(t-1)}$ and $r^{(t)}$ to $\widetilde{\boldsymbol{h}}^{(t)}$, illustrated in Figure A.4, as the note between u and h.

The GRU model and the LSTM are very similar and it is difficult to say which generally performs better, though the GRU seems to use less CPU time compared to the LSTM and might even update faster [12].

A.3 Naive Bayes

NB Classifiers use Bayes Theorem, where it is assumed that every feature is class-conditionally independent [36]. Conditionally independent means that the probability of some some feature F_i occurring given some class C, is not changed given another F_j, where $i \neq j$, written formally as

$$p(F_i \mid C, F_j) = p(F_i \mid C).$$

Though this assumption is not always applicable, NB still achieves comparable results to that of NN classifiers [36]. The probability of one class occurring given some features $F_1 \dots F_n$ is

$$p(C \mid F_1, \dots, F_n),$$

can be rewritten to

$$\frac{1}{Z} p(C) \prod_{i=1}^{n} p(F_i \mid C), \tag{A.4}$$

where $Z = p(F_1, \dots, F_n)$ and is therefore a constant[2]. A common way to pick the predicted class is choosing the most probable. Given some features $f_1, \dots, f_n$, it is possible to calculate the most probable as

$$\underset{c}{\operatorname{argmax}}\, p(C = c) \prod_{i=1}^{n} p(F_i = f_i \mid C = c).$$

[2]The formula can be derived using Bayes Theorem and the chain rule[36].

As this way of predicting means the NB model does not need to be very well estimated, but just well enough such that the correct class is more probable than the others.

A.4 Keyword extraction

Part of the solution presented in this report relies on the extraction of keywords from a corpus of opinion-rich texts. There are several approaches to do this, and this section will cover the underlying principles of some of these approaches.

A.4.1 Feature vector-based keyword extraction

A common way of extraction keywords is to extract words based on their statistical properties in the text. One way to facilitate this kind of extraction is to represent the documents in a corpus as a feature vector. This is formally referred to as the vector space model [45, Chapter 2]. The model consists of constructing a collection of document vectors in a matrix. Given a set of documents $D = \{d_1, d_2, \ldots, d_I\}$, a matrix M can be constructed such that the columns of the matrix each represent a unique word in the set of documents. Formally, the columns represent the set of words $W = d_1 \cup d_2 \cup \ldots d_J$. The rows in the matrix represent the feature vector of a document, meaning that a row in M is a vector v in which a weight is assigned for each word $w \in W$. As such, given a set of documents $D = \{d_1, d_2, \ldots, d_I\}$ and a set of unique words $W = d_1 \cup d_2 \cup \ldots d_J$, each cell $m_{nk} \in M_{I \times J} | 1 \leq n \leq I$, and $1 \leq k \leq J$ is assigned a weight that represents some statistical feature that is typically related to the frequency of $w_n \in d_k$.

A simple way to weight elements in M is to simply represent whether the word is present in the document by assigning the weight as a binary value; 1 if it is present, otherwise 0. However, one way to better ensure that keywords are captured (since keywords are likely to be central and frequent terms in all documents) is to weight elements in M according to the absolute frequency of the word in a document.

A problem with absolute frequency weighting is that a single document can contain a word with a very frequent word, misrepresenting the word as a keyword. This issue can also be encountered in longer documents. In order to combat this problem, we can weight elements of M regarding to their relative Term Frequency (TF); the absolute frequency of a word w in d normalised to the largest frequency of any word $v \in d$:

$$TF(w,d) = \frac{frequency(w,d)}{Max(\{frequency(v,d) | v \in d\})}$$

As such, TF gives us an idea of which words are frequent in all documents. However, there are many frequent words such as `"and"`, `"I"`, and `"but"` that do not afford any significant meaning or carry any importance to the central themes of the text. In order to capture the importance of a word in the corpus, we can use TFIDF. First, we need to define how important a word is a document, or rather if it is used frequently in all documents or not. This can be regarded as the logarithmically scaled total number of documents in D divided by the number of documents that contain the word. We will refer to this value as the Inverse Document Frequency (IDF) of a word $w \in W$:

$$IDF(w,D) = log\left(\frac{|D|}{|\{d|d \in D \text{ and } w \in W\}|}\right)$$

Combining the IDF of a word $w \in W$ and the term frequency of w in a specific document d, we acquire a notion of how important w is in d. This value is referred to as the TFIDF:

$$TFIDF(w,d,D) = TF(w,d) \cdot IDF(w,D)$$

Since TF is a direct notion of the frequency of w and IDF is a notion of the importance of w, the resulting value of TFIDF will yield a value representing whether a word w is both frequent and of high importance. Note that if w is a very common word, the ratio $\frac{|D|}{|\{d|d \in D \text{ and } w \in W\}|}$ approaches 1. Because of the logarithmic scaling, this will result in the value of TFIDF approach 0. Conversely, if w is not a common word across all documents, but is frequent within the current document, the IDF will approach 1, resulting in a high TFIDF value.

A.4.2 Graph-based keyword extraction

Instead of representing a corpus as a vector space as in subsection A.4.1, one could employ an entirely different way of thinking about the text in the corpus; namely by reasoning about the relations between words instead. This can be done by constructing a graph representing the words in the corpus and the relations between the words in individual documents. Instead of determining the importance of a word given a set of statistical features, we will determine the importance of a word by calculating the importance, or centrality, of a node in the graph representing the word. We will refer to words and documents using the same notation as in subsection A.4.1.

A graph $G = (N,E)$ consists of a set of nodes N and a set of edges E connecting two nodes $(u,v) \in E$ where $u,v \in N$. In one approach described in [24], each document $d \in D$ is processed in the usual reading order. First, a node is created for every word $w \in W$, yielding a set of nodes N. Then, while processing d in the usual reading order, an edge is established between two nodes $u,v \in N$ if they co-occur within a window of arbitrary length that slides along the document. For example, given the document `"graph theory is a fun subject"` and a window size of 3, then $N = \{\texttt{graph}, \texttt{theory}, \texttt{is}, \texttt{a}, \texttt{fun}, \texttt{subject}\}$, and after the first window pass, $E = \{(\texttt{graph, theory}), (\texttt{theory, is}), (\texttt{graph, is})\}$. In order to capture the order of the words in d, one could create edges between two words $u,v \in d$ if they co-occur in a left-to-right sequence such that u comes before v. Similarly to the vector space model, these bidirectional relationships can capture frequency-dependent features that can be calculated using TF or TFIDF.

When the graph has been built, the problem of determining which nodes are the most important. We can facilitate this by calculating centrality measures for the nodes in the graph.

Centrality measures

The centrality of a node is a value representing some measure of centrality for that node in relation to all other nodes in the graph and the graph structure. There are many centrality measures to choose from, and we will describe some of them in the following.

An easily calculated centrality measure is degree centrality:

$$Degree(n,G) = |\{e|e \in E \text{ and } n \in e\}|$$

Intuitively, the degree centrality simply represents how many nodes a single node is connected to by an edge. This centrality measure can be a good representation of central words in a text, but like TF described in subsection A.4.1, common words that do not carry much information can be over-represented here.

Instead, we can look at how close a node n is to every other node in the graph. This can be calculated as the inverse of the "farness" of the node; the average distance (length of the shortest path) from n to a node $u \in N$:

$$Closeness(n,G) = \frac{1}{\sum distance(n,u)|u \in N}$$

In terms of closeness, a node has a high closeness value if it is close to all other nodes in the graph. Intuitively, this is a decent representation of the centrality of a node - we assume a central node to be in the center, i.e. as close to everything else as possible.

Another approach for calculating the centrality of a node is by using PageRank [39]. PageRank is an algorithm designed to rank web-pages according to their importance. To intuitively understand the algorithm, consider a random surfer on the web. The surfer starts at a random web-page and takes one of the outgoing links on the webpage at random to continue to the next web-page. As such, at any web-page with L outgoing links, the surfer will take any one of these links with probability $1/L$. In order to not get stuck at a web-page with no outgoing links, the surfer will, at any node with no outgoing links, teleport to a random node in the graph. Furthermore, the surfer will also teleport from a node with outgoing links to a random node with probability α, and will continue the random walk (i.e. take a random outgoing link from the current node) with probability $1-\alpha$ where α is a constant value such that $0 < \alpha < 1$. The PageRank of a web-page ends up being a value between 0 and 1 that represents the fraction of steps the surfer spends at the page within some time limit [33, p. 464-465].

While we will not describe in detail how the algorithm works, the following presents a superficial description of PageRank and how it can be implemented to rank nodes in a graph.

The surfer is described by a probability vector q and a transition probability matrix P. Consider the web-graph in Figure A.5.

We define q^0 as the initial probability distribution. This vector represents the probability of transitioning to any of the web-pages in the graph from the current page. Say our surfer starts at A in Figure A.5. Then, as the initial probability distribution vector, q^0 becomes $(P(A), P(B), P(C), P(D)) = (1,0,0,0)$.

In addition to q, we have the transition probability matrix:

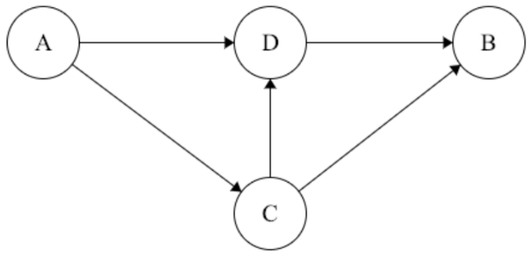


Figure A.5: A simple webgraph consisting of four web-pages A, B, C and D.

$$P = \begin{bmatrix} x_{1,1} & \dots & x_{1,j} & \dots & x_{1,i} \\ x_{i,1} & \dots & x_{i,j} & \dots & x_{i,n} \\ & & \dots\dots\dots\dots & & \\ x_{n,1} & \dots & x_{n,j} & \dots & x_{n,n} \end{bmatrix},$$

where $p_{i,j}$ is the probability of transitioning from page i to page j which, if there is a link from i to j, is equal to $\frac{1}{outdegree(i)}$.

In our example, the transition probability matrix is the following:

$$P = \begin{bmatrix} 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

In order to add the teleportation ability to the surfer, we replace all zero-cells of P with $\alpha \cdot \frac{1}{N}$ where N is the number of web-pages. To all other probabilities in P, we scale them by the probability of *not* taking the teleport and add $\alpha \cdot \frac{1}{N}$:

$$P = (1-\alpha) \begin{bmatrix} 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 1 & 0 & 0 \end{bmatrix} + \alpha \begin{bmatrix} 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{bmatrix}$$

Now, we are ready to determine the PageRank of the web-pages. We use power iterations to iteratively calculate a new probability vector $q = q \cdot P$. We iterate in time steps, so at any given time t, we can determine the next probability vector: $q^{t+1} = q^t P$. We continue these iterations until q converges. The resulting probability vector q is a vector containing the PageRanks of the web-pages in our graph.

There are many more centrality measures that we could try to apply, such as HITS [33, p. 478]. For all the different measures, they all share the common goal of determining how central a node is - what varies is the notion of when a node is central in a graph. In closing, graph-based keyword extraction approaches have proven to be both effective in terms of accuracy and validity of the extracted keywords and in terms of running times, as has been shown with the TKG approach devised by [2]. We will base our graph-based approach for KE on TKG in subsection 5.8.2.

A.5 Accuracy measures

In order to evaluate our SA and KE models we need to determine how we will measure their accuracy. There are several ways this can be done, but some measures that are commonly used in this field are accuracy, precision and recall [34]. We will describe how the measurements are calculated and what information is gained from using them on a model.

For classification tasks such as SA the accuracy can be measured simply by taking the percentage of results which were correctly classified. Though this works well for some tasks there are some issues which can make it unsuitable in some cases. One issue is that this metric doesn't give much insight into what the composition of results is. Take a balanced binary classification problem; an accuracy of 70% could signify that each item has 70% chance of being classified correctly or it could mean than the model classifies all items in one category correctly but only 40% of the other category's items. This might be perfectly acceptable in some cases but a problem in others. The distribution of results might thus reveal patterns that can help to suggest modifications to improve the model.

In order to evaluate the accuracy of a model in a more meaningful way, two useful metrics are to measure the ratio of the returned results which are correct as well the ratio of all correct results which are returned, which are called the precision and recall respectively.

That is, given a dataset, D, and a model, M, that extracts some information(e.g. keywords), $M(D)$, from this dataset. Let T_D be the set of all information that should be extracted from D, F_D be the set of all information that shouldn't such that $F_D = \{x | x \notin T_D\}$. The set of correct extracted information, $C_{M(D)}$ is then the intersection between extracted information and correct information, that is $C_{M(D)} = M(D) \cap T_D$. A visualisation of the different sets can be seen in Figure A.6.

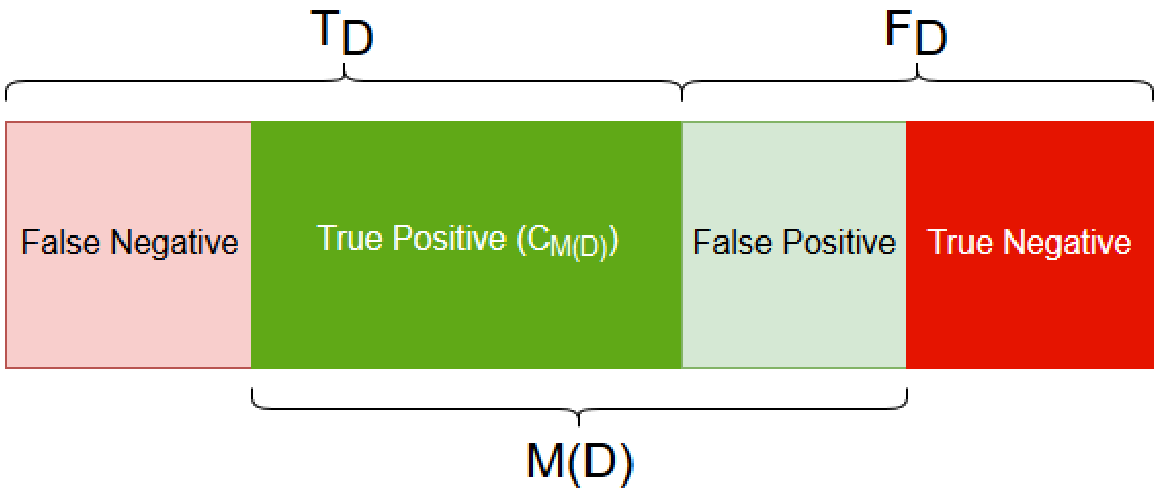


Figure A.6: Visualization of the values used to calculate precision and recall for a binary decision.

Precision and recall is then calculated as follows:

Precision is $P_{M(D)} = \frac{|C_{M(D)}|}{|M(D)|}$.

Recall is $R_{M(D)} = \frac{|C_{M(D)}|}{|T_D|}$.

It is worth noting that neither precision nor recall take the true negatives (the right-most block in Figure A.6) into account. To see that this can be an issue consider a binary decision case where a model given a input dataset with 1000 items should return true for 10 of them and false for the remaining 990. If the model gave 1 true positive and 9 false positives then precision and recall would both be 10% which might seem bad, but the model had in fact given the correct value, false, in $>$ 99% of the cases.

This shouldn't be an issue in our case due to the nature of our analytic tasks. With regards to keyword extraction the "true negative" category encompasses every word which is not relevant as a keyword for the given text, a group which is orders of magnitude larger than the amount of relevant keywords. To take this group into account in our accuracy evaluation would thus likely overshadow the distribution of the far smaller groups of relevant and found keywords.

With regards to our sentiment extraction, it should also not be a significant issue due to using a balanced dataset. The accuracy relative to negatives is thus inferable from the precision and recall measures.

A.5.1 F-measure

It is desirable to combine the precision and recall measures into a single score. While one could simply take their average this is not ideal as one could always get an average >50% simply by classifying everything as positive, thus ensuring a recall of 1. To let the combined score more useful we can use the harmonic mean to derive a F-measure, as seen in Equation A.5.1. This measure will lean more towards the smaller measure thus giving a better indication of a models performance.

$$F = \frac{2 \times precision \times recall}{precision + recall} \tag{A.5}$$

While this measure is better, it might still be inadequate for some applications where either precision or recall is more significant than the other. We can thus instead use a weighted harmonic mean which we can reduce to to the following:

$$F_\beta = \frac{1}{\alpha \times precision^{-1} + (1 - \alpha) \times recall^{-1}} \tag{A.6}$$

$$= (1 + \beta^2)\frac{precision \times recall}{(\beta^2 \times precision) + recall} \text{ where } \beta = \frac{1 - \alpha}{\alpha} \tag{A.7}$$

In Equation A.7 β is a measure of how important recall is relative to precision, with lower values (eg. $\beta < 1$) weighing precision more and higher values (eg. $\beta < 1$) weighing recall more.

Chapter B: Server setup

To simulate our system we utilised a containerised approach using Docker and docker-compose. Docker and docker-compose allow us to run multiple virtual machines which are isolated from the host machine yet still connected to each other using virtual networks. These virtual networks are simulating the connections between the containers that make up the system. Each separate container is simulating a service in a deployed system where each service is assumed to be deployed on an isolated physical server.

As described in section 5.4 our APIs are RESTful which is achieved through the use of Python with the Flask framework. Flask itself, however, is not enough in itself to provide a stable hosted service, for a production environment, which means additional setup will be required on the server side of the system. This additional setup is not defined as part of the system as it would limit the system to additional technologies that can be avoided. In our development environment we used Gunicorn as our choice of Web Server Gateway Interface (WSGI) which, contrary to Flask, is able to handle the stress of a production environment. The purpose of the WSGI is to relay the incoming requests to the API and queue requests while the service is busy with other requests. The WSGI also allows the API to serve multiple requests at once and, based on our tests, make sure that no request is being starved by others.

In our development environment we used only one container, or server, for our database. This may or may not be different in an actual production environment based on factors such as, additional cost of having multiple servers for different database systems as well as stability as other databases would not become unresponsive in case a database server went down were they split on multiple physical servers.

The cooperation between the different services are illustrated in Figure B.1. The dotted arrows indicate that a service is using the service pointed to and a filled arrow indicates which virtual network a service is using. The squares are docker containers, here simulating servers of the system, and the quintuples are virtual networks. Do note that the virtual networks are purely made as a step to make the development setup easier for us to modify and maintain.

The client container is where the UI is hosted and will be the entrypoint for the system from a users perspective, this service will be used to authenticate with the system. The gateway API is what links the client to the rest of the system by providing the client with the requested data. In order to provide this data, several other services are used. The information service is responsible for gathering information and running it through SA and KE. When the data has been processed it is sent to the database service so it can be accessed by other services. When the client requests to view statistics for a brand, the data is run through the statistics service which provide the last step of the data processing.

Besides the services we provide, our system is dependent on the service called tf-serving (TensorFlow Serving) and is a service provided by google, that we host on the server alongside our own services, with which we host our SA model used by the SA service as illustrated by the closed network tensorflow on Figure B.1.

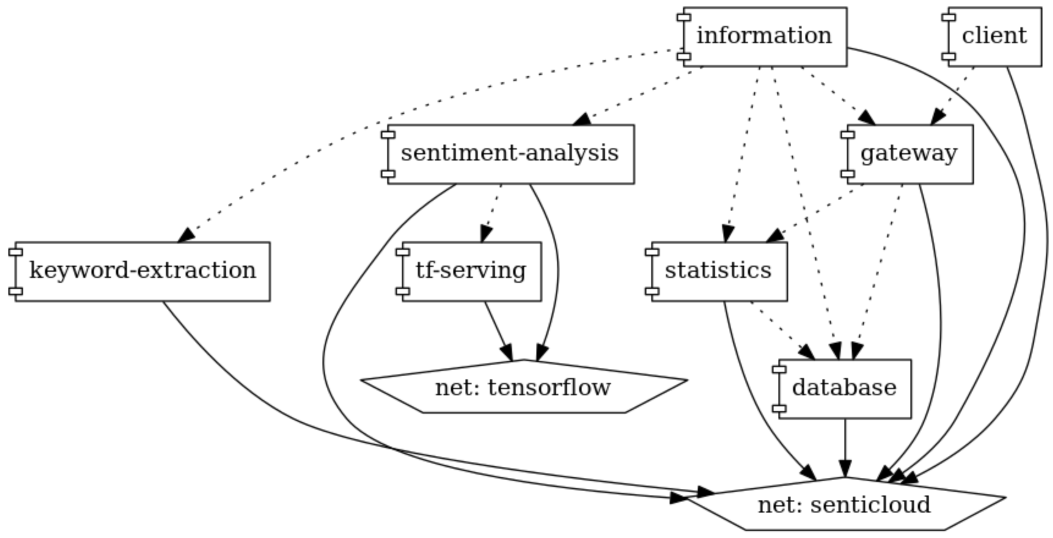


Figure B.1: Visualisation of the containerised system on our development server simulating the communication between the different services.

Chapter C: Usability test assignment

Bob Andersen is the marketing manager at Jagex Limited, a British video game company. Bob has a hard time figuring out if his marketing campaigns have a positive or negative effect on the company's reputation. To do something about that, Bob decides to try SentiCloud.

Bob arrives on SentiCloud's website and now has to figure out how to create a user account such that he can log in and start using the platform.

1. Create a user account.
2. Log in with the newly created user account.

Once logged in, Bob wants to start monitoring the company's reputation as well as its most used product.

3. Add Jagex as a brand.
4. Add Runescape as a brand.
5. Add the synonym osrs to Runescape.

After adding the brands, the system begins collecting data. Bob returns to the system after a week, to see what SentiCloud has found on his brands.

6. Find the top keywords for Jagex at a given time.
7. Figure out if the posts mentioning Jagex the past week have a positive or negative trend.

Bob started a new marketing campaign for Jagex last wednesday and now wants to see if the marketing campaign had an impact on the company's reputation.

8. In the settings for the Jagex page set the date range to last wednesday to friday and the granularity to hour.
9. Determine the sentiment peak value.

Bibliography

[1] W. M. a, A. Hassan, and H. Korashy, “Sentiment analysis algorithms and applications: A survey”, volume 5, 2014 (cited on page 45).

[2] W. D. Abilhoa and L. N. de Castro, “A keyword extraction method from twitter messages represented as graphs”, 2014 (cited on pages 49, 51–53, 95, 118).

[3] *Accurate opinion data through advanced sentiment analytics*, (accessed 24-09-2018). [Online]. Available: `https://www.brandseye.com/` (cited on page 8).

[4] *Announcing ‘machine learning .net’ 0.5*, `https://blogs.msdn.microsoft.com/dotnet/2018/09/12/announcing-ml-net-0-5/` (cited on page 32).

[5] O. Araque, I. Corcuera-Platas, J. F. Sánchez-Rada, and C. A. Iglesias, “Enhancing deep learning sentiment analysis with ensemble techniques in social applications”, *Expert Systems with Applications*, volume 77, pages 236–246, 2017. DOI: `10.1016/j.eswa.2017.02.002` (cited on page 46).

[6] A. B. Bondi, “Characteristics of scalability and their impact on performance”, in *Proceedings of the 2nd international workshop on Software and performance*, ACM, 2000, pages 195–203 (cited on page 12).

[7] A. H. Brams, A. L. Jakobsen, M. R. Jensen, S. R. Hansen, T. E. Jendal, and T. B. Andersen, “Dice: A step forward for astep - driver identification and count estimation with deep learning trajectory analysis”, 2018 (cited on page 68).

[8] E. Cambria, B. Schuller, B. Liu, H. Wang, and C. Havasi, “Knowledge-based approaches to concept level sentiment analysis”, 2013 (cited on page 44).

[9] K. Cho, B. v. Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, Y. Bengio, and D. Bahdanau, *Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Sep. 2014. [Online]. Available: `https://arxiv.org/abs/1406.1078` (cited on page 113).

[10] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, *On the properties of neural machine translation: Encoder-decoder approaches*, Oct. 2014. [Online]. Available: `https://arxiv.org/abs/1409.1259` (cited on pages 112, 113).

[11] chrisdavidmills, *Cross-origin resource sharing (cors)*, Dec. 2018. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS` (cited on page 36).

[12] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, *Empirical evaluation of gated recurrent neural networks on sequence modeling*, Dec. 2014. [Online]. Available: `https://arxiv.org/abs/1412.3555` (cited on pages 109, 112, 113).

[13] *Comparison with other frameworks*, https://vuejs.org/v2/guide/comparison.html (cited on page 31).

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (Second ed.)* MIT Press, 2001 (cited on page 87).

[15] M. J. Culnan, P. J. McHugh, and J. I. Zubillaga, "How large u.s. companies can use twitter and other social media to gain business value", 2010 (cited on page 1).

[16] L. Dey, S. Chakraborty, A. Biswas, B. Bose, and S. Tiwari, "Sentiment analysis of review datasets using naïve bayes' and k-nn classifier", *International Journal of Information Engineering and Electronic Business*, volume 8, pages 54–62, Jul. 2016. DOI: 10.5815/ijieeb.2016.04.07 (cited on page 45).

[17] J. L. Elman, "Finding structure in time", *Cognitive Science*, volume 14, number 2, pages 179–211, 1990. DOI: 10.1207/s15516709cog1402_1 (cited on page 109).

[18] P. Gamallo and M. Garcia, "Citius: A naive-bayes strategy for sentiment analysis on english tweets", in *Proceedings of the 8th international Workshop on Semantic Evaluation (SemEval 2014)*, 2014, pages 171–175 (cited on page 45).

[19] M. A. Garcıa-Cumbreras, M. G. Vega, F. M. Santiago, and J. M. Peréa-Ortega, "Sinai at weps-3: Online reputation management", 2010 (cited on page 1).

[20] P. Gonçalves, M. Araújo, F. Benevenuto, and M. Cha, "Comparing and combining sentiment analysis methods", 2013 (cited on pages 7, 44–46).

[21] A. Hassan and A. Mahmood, "Deep learning approach for sentiment analysis of short texts", pages 705–710, Apr. 2017. DOI: 10.1109/ICCAR.2017.7942788 (cited on pages 45–47).

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural Computation*, volume 9, number 8, pages 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735 (cited on page 110).

[23] A. Hulth, "Improved automatic keyword extraction given more linguistic knowledge", 2003 (cited on page 85).

[24] W. Jin and R. K. Srihari, "Graph-based text representation and knowledge discovery", 2007 (cited on page 115).

[25] T. Jost, A. Dosovitskiy, T. Brox, and M. Riedmiller, *Striving for simplicity: The all convolutional net*, Apr. 2015. [Online]. Available: https://arxiv.org/abs/1412.6806 (cited on page 47).

[26] Y. Kim, *Convolutional neural networks for sentence classification*, Sep. 2014. [Online]. Available: https://arxiv.org/abs/1408.5882 (cited on page 108).

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", *NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems*, volume 1, pages 1097–1105, Dec. 2012. DOI: 10.1145/3065386 (cited on page 107).

[28] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, volume 86, number 11, pages 2278–2324, 1998. DOI: 10.1109/5.726791 (cited on pages 107, 108).

[29] J. Lewis and M. Fowler, *Microservices: A definition of this new architectural term*, Accessed 4 Oct. 2018. [Online]. Available: `https://martinfowler.com/articles/microservices.html` (cited on page 28).

[30] Z. C. Lipton, J. Berkowitz, and C. a. Elkan, *A critical review of recurrent neural networks for sequence learning*, Oct. 2015. [Online]. Available: `https://arxiv.org/abs/1506.00019?fbclid=IwAR2lvd7MSWZEQyMGN-QEPvLZpAwG33ufsGt_bMFMwPLhjU8Lezbv5QbxuEA` (cited on pages 109, 110).

[31] B. Liu, "Sentiment analysis and opinion mining", *Synthesis Lectures on Human Language Technologies*, volume 5, pages 1–167, 2012. DOI: `10.2200/s00416ed1v01y201204hlt016` (cited on page 1).

[32] *Liwc*, `http://liwc.wpengine.com/how-it-works/`, Accessed 17 Oct. 2018 (cited on page 46).

[33] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. 2008 (cited on pages 116, 117).

[34] ——, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008, ISBN: 0521865719, 9780521865715 (cited on page 118).

[35] I. Miri, "Microservices vs. soa", Jan. 2017. [Online]. Available: `https://dzone.com/articles/microservices-vs-soa-2` (cited on page 29).

[36] M. N. Murty and V. S. Devi, *Pattern recognition: an algorithmic approach*. Springer, 2012 (cited on page 113).

[37] myrdd, *Server-side access control (cors)*, Apr. 2018. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Server-Side_Access_Control` (cited on page 36).

[38] J. Nielsen, *Usability Engineering*. 1993 (cited on page 9).

[39] L. Page and S. Brin, "The anatomy of a large-scale hypertextual web search engine", 1998 (cited on page 116).

[40] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up? sentiment classification using machine learning techniques", 2002 (cited on page 45).

[41] *Programming language benchmarks*, `https://github.com/kostya/benchmarks` (cited on pages 32, 33).

[42] *Python's web framework benchmarks*, (accessed 19-11-2018), 2016. [Online]. Available: `http://klen.github.io/py-frameworks-bench/` (cited on page 34).

[43] *Results for js web frameworks benchmark*, `https://www.stefankrause.net/js-frameworks-benchmark6/webdriver-ts-results/table.html` (cited on page 31).

[44] C. Richardson, *What are microservices?*, Accessed 4 Oct. 2018. [Online]. Available: `microservices.io` (cited on pages 28, 29).

[45] P. Sacks, *Techniques of Functional Analysis for Differential and Integral Equations*. 2017 (cited on page 114).

[46] C. N. dos Santos and M. Gatti, "Deep convolutional neural networks for sentiment analysis of short texts", 2014 (cited on page 45).

[47] S. Tan, X. Cheng, Y. Wang, and H. Xu, “Adapting naive bayes to domain adaptation for sentiment analysis”, *Lecture Notes in Computer Science Advances in Information Retrieval*, pages 337–349, Apr. 2009. DOI: `10.1007/978-3-642-00958-7_31` (cited on page 48).

[48] Y. R. Tausczik and J. W. Pennebaker, “The psychological meaning of words: Liwc and computerized text analysis methods”, *Journal of Language and Social Psychology*, volume 29, 2011 (cited on page 44).

[49] M. Thelwall, *The Heart and Soul of the Web? Sentiment Strength Detection in the Social Web with SentiStrength*. 2017 (cited on page 45).

[50] M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas, “Sentiment strength detection in short informal text”, *Journal of the American Society for Information Science and Technology*, volume 61, number 12, pages 2544–2558, 2010. DOI: `10.1002/asi.21416` (cited on page 46).

[51] Y. Uzun, “Keyword extraction using naive bayes”, 2005 (cited on page 49).

[52] W. Yin, K. Kann, M. Yu, and H. Schütze, *Comparative study of cnn and rnn for natural language processing*, Feb. 2017. [Online]. Available: `https://arxiv.org/abs/1702.01923?fbclid=IwAR0PWOuyWYy8_ganjAHK8b9A5TwIJ51UcT7VxTuUGtK4TccqsQ7N5Y_gNh0` (cited on pages 46, 47, 108).

[53] T. Young, D. Hazarika, E. Cambria, and S. Poria, *Recent trends in deep learning based natural language processing*, Nov. 2018. [Online]. Available: `https://arxiv.org/abs/1708.02709` (cited on page 46).

[54] J. Zabin and A. Jefferies, *Social media monitoring and analysis: Generating consumer insights from online conversation*, Aberdeen Group Benchmark Report, Jan. 2008 (cited on page 1).

Glossary

API Application Programming Interface. 11, 20, 22, 24, 27, 28, 30, 32, 34–36, 41, 43, 55–57, 61, 62, 64, 65, 67, 68, 90–92, 100–102, 105, 121

BPTT Backpropagation Through Time. 109

CI Continuous Integration. 98
CNN Convolutional Neural Network. 45, 47, 48, 68, 83–85, 94, 95, 107, 108
CORS Cross-Origin Resource Sharing. 36
CRUD Create, Read, Update, Delete. 43, 79
CSR Client-Side Rendering. 27

DBMS Database Management System. 13, 20, 37, 40–42, 76, 78, 79, 81, 92–94, 100, 102, 105
DNN Deep Neural Network. 46

FC Fully Connected Neural Network. 107

GRU Gated Recurrent Unit. 46, 47, 68, 83, 112, 113

IDF Inverse Document Frequency. 114, 115

KE Keyword Extraction. 1, 9, 13, 17, 19, 26, 32, 33, 35, 38, 48, 49, 52, 57, 59–62, 75, 83, 85, 90–96, 100, 102, 103, 105, 107, 118, 121
KG Keyword Graph. 35, 49, 51–53, 69, 71, 85, 89, 90, 95

LIWC Linguistic Inquiry and Word Count. 44–46
LSTM Long Short Term Memory. 45–47, 110, 112, 113
LSVRC Large Scale Visual Recognition Challenge. 107

ML Machine Learning. 7, 9, 32, 33, 35, 44–46, 48, 49, 94, 107
MPA Multiple-Page Application. 29
MVC Model-View-Controller. 30, 32

NB Naïve Bayes. 45, 48, 49, 66, 84, 95, 113, 114
NLP Natural Language Processing. 7, 46
NN Neural Network. 45, 48, 84, 95, 113

ORM Object-Relational Mapping. 20, 42, 55
OS Operating System. 99

REST Representational State Transfer. 92
RNN Recurrent Neural Network. 46, 47, 68, 83, 85, 95, 108–110
RPS Requests Per Second. 76–78, 80

SA Sentiment Analysis. 1, 7, 8, 13, 17, 19, 26, 33, 35, 38, 44, 46, 55, 57, 59–61, 66, 68, 75, 83, 84, 90, 91, 94, 102, 103, 105, 118, 121, 122
SMP Social Media Post. 1, 7, 8, 11, 12, 18, 38–41, 48, 49, 51, 71, 92, 93, 95, 99–102, 105
SPA Single-Page Application. 29, 30, 35
SSR Server-Side Rendering. 27

TDD Test-Driven Development. 4, 14, 98, 103
TF Term Frequency. 114–116
TFIDF Term Frequency-Inverse Document Frequency. 49, 50, 69, 70, 85–90, 95, 96, 114, 115
TKG Twitter Keyword Graph. 51, 95, 118

UI User Interface. 26, 27, 30, 31, 55, 82, 101–103, 121
URL Uniform Resource Locator. 24, 25, 58

WAF Web Application Framework. 30, 31
WSGI Web Server Gateway Interface. 121