Självständigt arbete på grundnivå

Independent degree project - first cycle

Datateknik
Computer Engineering

Improving the Chatbot Fallback Experience
With a Content-Based Recommender System

Angelica Gardner

Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MID SWEDEN UNIVERSITY
Department of Information Systems and Technology

Examiner: Mikael Hasselmalm, mikael.hasselmalm@miun.se
Supervisor: Mattias Dahlgren, mattias.dahlgren@miun.se
Company supervisor: Alaa Alweish, aal@simplea.com
Author: Angelica Gardner, anga1306@student.miun.se
Degree programme: Web Development, 120 hp
Main field of study: Computer Engineering
Semester, year: Spring, 2019

Abstract

Chatbots are computer programs with the capability to lead a conversation with a human user. When a chatbot is unable to match a user's utterance to any predefined answer, it will use a fallback intent; a generic response that does not contribute to the conversation in any meaningful way. This report aims to investigate if a content-based recommender system could provide support to a chatbot agent in case of these fallback experiences.
Content-based recommender systems use content to filter, prioritize and deliver relevant information to users. Their purpose is to search through a large amount of content and predict recommendations based on user requirements. The recommender system developed in this project consists of four components: a web spider, a Bag-of-words model, a graph database, and the GraphQL API.
The anticipation was to capture web page articles and rank them with a numeric scoring to figure out which articles that make for the best recommendation concerning given subjects. The chatbot agent could then use these recommended articles to provide the user with value and help instead of a generic response.
After the evaluation, it was found that the recommender system in principle fulfilled all requirements, but that the scoring algorithm used could achieve significant improvements in its recommendations if a more advanced algorithm would be implemented. The scoring algorithm used in this project is based on word count, which lacks taking the context of the dialogue between the user and the agent into consideration, among other things.

Keywords: Chatbot, Recommender system, Web crawling, Bag-of-words, Graph database, GraphQL

Acknowledgements

First and foremost, I would like to thank God without whom nothing would be possible.
Then I would like to extend my gratitude and dedicate a big thank you to several people whose help has been of great value during this project work.

Appreciation is directed to my examiners; Mattias Dahlgren and Mikael Hasselmalm. They have followed my classmates and me through the Web Development programme at Mid Sweden University, providing us with support and inspiration. If there's anything I would say I've benefited from, it's the knowledge I've gained in different areas of web development while also staying up-to-date with new techniques and trends.
A big thank you to all other teachers and lecturers we've had during these years that have given their portion of useful information and feedback.

During this independent degree project, I've done an internship at [A]. At [A], an inclusive language is used and to honour that commitment; my sincerest gratitude is expressed to the whole team and everyone I've spoken to at [A] during this time, without mentioning individuals. No one mentioned, no one forgotten. As I've learned from [A], solving problems is a team effort.

Last but not least, I would like to extend my sincerest gratitude to the people close to me who have helped me in any matter and who have shared my effort to make this project a reality. This is especially true for my husband, Najib. You're always around with unfailing support; inspiring me with your creativity, pushing me to think outside the box and to break out of my comfort zone. I want to let you know how much I greatly value your contribution and sincerely appreciate your belief in me. No amount of words could ever describe how grateful I am to you.

Angelica Gardner
Stockholm, June 2019

Table of Contents

Terminology

Below are explanations for abbreviations that appear in the essay.
They are presented in alphabetical order.

Acronyms/Abbreviations	**Meaning**
BoW	Bag-of-words
Fallback intent	A situation that happens when a chatbot agent is not able to match a user's utterance to any predefined response
FAQ	Frequently Asked Questions
HTML	Hypertext Markup Language
IDE	Software application that provides facilities to programmers for software development. An IDE normally consists of at least a source code editor, build automation tools, and a debugger.
Intent	The need and intention of a user
JSON	JavaScript Object Notation
NLU	Natural Language Understanding
Use case	Describes one specific interaction between a human user and the software.
Webhook	A way for an app to provide other applications with real-time information.

1 Introduction

This chapter begins with a background and a problem statement, followed by the purpose and scope of this project. In the end, the chosen disputation is presented.

1.1 Background and problem motivation

Chatbot technology is an invention where a computer program has the capability to lead a sensible, conversational dialogue with a human user - acting independently (i.e. without human supervision) as customer support, answering questions and providing information. Chatbot technology is being used in industries like e-commerce, booking sites, service solutions, and education - and just last year Forbes [1] published a piece about the growing trend on how companies are increasingly using chatbots to boost their business and how these applications are capable of handling more and more advanced cases of user contact.

However, the design, development and maintenance of a chatbot system is not without a hassle. [3] One feature that is common to all approaches of building chatbots is that the development process is time-consuming and far from straightforward. When setting up a chatbot application, you have to create a large number of rules and subjects the chatbot agent needs to know about to ensure that the program can lead a proper dialogue with human users.
Another difficulty is to keep track of user intent (i.e. what the user wants with his/her statement) because utterances are often dependent on the context in which they were given. This is especially difficult when the user provides an unexpected input that does not match any of the rules and subjects that have been set up.

One way chatbots handle this situation is through “fallback intent”. A fallback intent [2, Default intents] is a kind of catch-all for any unrecognized user input, like out-of-domain requests or unexpected statements. When these situations happen, a chatbot application will have a set of predefined generic responses that its agent can use as an answer when it couldn’t match the input to anything else.
The dilemma is that these generic responses don’t contribute to the conversation in any meaningful way - they might even cause the dialogue to discontinue while providing the user with a bad experience. The reality is that today, users expect far better than explanatory error messages. [4]

There are some strategies chatbot developers use to sustain engagement with their users in case of fallback intent like rotating between multiple fallback messages,

showing related FAQ articles, asking follow-up questions, sending context reminders, or even providing the opportunity to contact a human.
Regardless of which strategy that's chosen, a user's engagement can't be automatically generated, it stems from how useful your chatbot is to the user. [3] From this, we can understand that the chatbot fallback experience needs to provide some value to the end user that helps him/her to find the answer he/she was looking for.

Another type of system application that has been doing well in the same industries as chatbot technology is recommender systems [5]. The purpose of a recommender system is to prioritize and efficiently deliver relevant information to a user. Many recommender systems use content-based filtering [6] when it is predicting recommendations. The system does this by using a scoring algorithm that takes into account the user's previous interactions with the content or other specified criteria.

1.2 Company introduction

This independent degree project is carried out as an internship at [A]. [A] is a Content Intelligence Service that partners with leading global enterprises. [A] orchestrates content intelligence systems that unify people, processes, and technology for omnichannel publishing and real-time personalised customer experiences at scale. [7, About]

1.3 Overall aim

This project aims to explore the possibility of improving the chatbot fallback experience with support from a content-based recommender system. By developing the recommender system as an independent module, the aim is for the application to support an already existing chatbot platform in case of fallback intent. This recommender system could also be used by the chatbot agent in case the user explicitly asks to be recommended some content asset about a mentioned topic.

Problem statements:

- Can a recommender system be used to improve the chatbot fallback experience?
- Will the recommender system's scoring algorithm suggest content that is relevant to the user's input?

1.4 Scope

The scope of this project will be to build a basic system application for a content-based recommender system that will have just enough functionality to produce some recommendations as a result. This way, it's possible to analyse and evaluate the system and answer the problem statements.
Ideas for improvements and future work is mentioned in the conclusions.

The recommender system will be built with a set of technologies chosen in consultation with my supervisor at [A]. Therefore, implementations in other programming languages, frameworks or technologies are out of scope of this project.

Due to time constraints that exist, some parts of the recommender system may be based on open source libraries and not developed from scratch.

1.5 Concrete and verifiable goals

The purpose of the recommender system is to gather articles from a website, rank the content concerning its topics and suggest the highest scoring article based on those rankings when a chatbot agent asks for recommendations that match a user's input.

For the system to be able to fulfil this purpose, the following technical requirements should be met. The system should:

- Contain a web spider (crawler and parser) that fetches and parses content from one website at a time.
- Rank the content for specific subjects to tell how relevant each piece of content is to any given topic.
- Store the content and its rankings in permanent storage.
- Grant API access to other system applications so that they can get and use the stored data.

Also, in consultation with [A], the following methodologies are expected to be followed to deliver the recommender system. The system should:

- Use .NET Core for backend development.
- Present data in a graph format, displaying nodes and relationships between the content.
- Use Dialogflow's API to synchronize the system's graph data with a chatbot agent.

1.6 Outline

The structure of this thesis will be as followed:

Chapter 2 consists of an introduction to the areas of knowledge that relate to the development of the recommender system. These areas include chatbot technology, recommender systems, web crawling and parsing, and graph databases.

Chapter 3 presents the method taken to fulfil the project’s purpose and answer the research questions.

Chapter 4 contains the decisions taken during the implementation and development of this system.

Chapter 5 presents the project result and its findings.

Chapter 6 discusses suggestions for future work and reflects on ethical considerations related to the use of the recommender system.

Chapter 7 presents the conclusions drawn from working with this project together with answering the research questions.

2 Background

This chapter provides an introduction to the technical areas of knowledge that relate to this report. In the first part of this chapter, an overview of chatbot technology and recommender systems will be presented. Then follows a description of what each part of the recommender system module is supposed to do. The chapter is concluded with some theory on how to evaluate the results.

2.1 Chatbot technology

Shevat [3] describes chatbot technology as software programs used to imitate human conversation in a text-based format through inputs from human users and outputs from a chatbot agent. We can see it as chatbots providing a way to expose users to software services through a conversational interface.

There are several types of chatbot systems, and they differ in complexity and architecture, but all kinds require the application to interpret and process the user's input in some way as well as for the chatbot agent to pick an answer to output. The response from the chatbot agent is usually selected from a set of predefined answers.

2.1.1 Natural language understanding

An increasingly popular mechanism to use when building chatbot engines are natural language understanding (NLU) tools. NLU is a powerful tool for processing and understanding user input when creating conversational user interfaces. [2, Overview] Chatbot frameworks that utilize NLU let the chatbot agent derive an intent (the need and intention of the user) from natural human language. The chatbot framework will also extract entities (conversational context variables) from that input.
Extracting these intents and entities is an essential aspect of conversation management when trying to keep track of the dialogue context.

As a chatbot application falls into fallback intent, the intent and entities that the agent has picked up from the conversation with the user can be of great importance to use as parameters for a recommender system.

2.2 Recommender systems

Recommender systems have the primary purpose of filtering, prioritizing, and efficiently deliver relevant information to internet users [5].

The systems do this by searching through an often large amount of content and filter out vital information according to users' preferences, interests, or observed behaviour. It will then predict whether a particular user would prefer this item or not based on a set of criteria.

Recommender systems have been proven to enhance both the decision-making process of users as well as how they perceive the quality of services. [5] The system supports users to move beyond standard and generic site searches that often lead to irrelevant content. [8] Search functionality use a broad match keyword strategy which will yield in a high search result volume, but the downfall is that results are often irrelevant and loosely linked to the context the user was looking for. Generic search results will rarely produce anything specific enough to make the searcher satisfied.

For a recommender system to solve this issue, it needs to use techniques that will provide users with actual relevant and trustworthy recommendations.

In this project, the most suitable approach to use when building the recommender system is content-based filtering.

2.2.1 Content-based recommender systems

Content-based recommender systems give support to the idea of using content for decision-making in the recommendation process. Content-based filtering techniques are the most successful type of recommender system techniques when the recommendations are concerning documents such as web pages, publications and news [6] because these types of filtering techniques heavily rely on ratings, recommendation scores, and patterns found in both the content item and the user request.

A content-based recommender system will need to go through a set of phases:

1. **Information collection phase.** The recommender system relies on input about users, so during this phase, the system will collect data needed for the prediction tasks.
2. **Learning phase.** The next step is to generate recommendation candidates the system thinks might be relevant to the user. Here, the system will apply a filtering algorithm to all of its items. Each item will be assigned a score depending on its features. During this phase, candidates might be filtered out if the recommendation score isn't high enough.
3. **Prediction phase.** During this phase, the system predicts what kind of item(s) the user may prefer through candidate ranking. Many candidates will appear as a match more than once, and this needs to boost their recommendation

score so that the total rankings can be combined. After that, it can be a matter of sorting the list of recommendations by score.
4. **Recommendation phase.** The result is handed over to be displayed to the user by whatever display layer being used.

One shortcoming of content-based filtering techniques is that they require the content to be organized and well-structured so that the system can evaluate which content pieces that fit as recommendation candidates. In the case of recommending web pages, this consideration needs extra attention because of the way web pages are built with HTML. HTML is not considered to be truly structured [9], and each website can have several different templates for how the source code should display the content.

2.3 Web spider

As mentioned, content-based recommender systems base their recommendations on content items and their features. This assumes that there are content items available for the system to use when comparing and evaluating, preferably stored in a connected database ready for querying.
Wang, D. et al. [6] suggest that a web crawler can be employed in a recommender system to update the set of content items continuously. This proposed approach of using a web crawler as a component in the recommender system would suit the purpose of this project as it’s supposed to provide support for a chatbot application used on websites.

2.3.1 Web crawling

A web crawler often referred to as a “spider”, is a type of bot that systematically visits and downloads web pages on the internet. [10] The spider will visit a website URL and start by downloading all HTML source code; it will extract the links it finds in the code and places these links in a queue. When crawling the document, the spider will also collect other information of interest from the HTML-code with the help of a web scraper.
When the spider is finished with one page, it will check the link queue for another URL to be visited, and the process is repeated.

2.3.2 Web scraping

Scraping involves fetching data from the web page that the crawler visits, and extract its information. Once the crawler fetches the page, the data can be processed and harvested by the scraper. [10]

There are several difficulties in identifying and classifying content on the internet today when the web is so extensive and has a lot of information that is mostly made up of semi-structured documents. [9] These challenges relate not only to the fact that each website might have a unique content structure (making it challenging to produce a generic scraper), but sites might also undergo structural changes quite often. Choosing a proper selection technique when scraping will be one of the critical aspects of this part of the recommender system.

2.3.3 Guidelines and policies

Most spiders are implemented according to guidelines and policies. These best practices exist for the spider to cope with both technical requirements and ethical aspects. [11] A spider that is developed without regards to these guidelines or policies may cause problems for the website and server it's visiting. These issues mainly relate to data privacy, copyrighted material, cost of bandwidth, and website performance.

Castillo [10] suggests that the behavior of a web crawler is the outcome of a combination of the following policies:

- **A selection policy.** This policy will state which web pages the spider is supposed to crawl and download by telling it which links to follow. This will ensure that the information is relevant to the user and not a random collection of pages.
- **A re-visit policy.** Content on the web can both change and become outdated at high speed, and this policy relates to that fact. If the organisation wants an updated index, it needs to take into consideration the need to re-visit already crawled pages to look for new, updated, or deleted data.
- **A politeness policy.** This policy tries to solve the issue that crawlers require a lot of resources from servers. Here, a crawler might be told to avoid visiting the pages of a website that the owner doesn't want to be crawled. The spider will also be told how many seconds to wait between each request.
- **A parallelisation policy.** A parallel spider is a crawler that runs multiple processes simultaneously in parallel. Each process manages a unique web page which is retrieved from a shared queue. The goal of this policy is to maximise the download rate while also avoiding repeated downloads of the same page.

In this project, some of these policies will be of greater importance than others.

2.4 Graph databases

In the introduction to chapter 2.3, it was mentioned that the recommender system would require some permanent storage, a connected database. In this project, we'll be taking advantage of a graph database.

A graph database is a type of NoSQL database that uses graph structures to represent and store the information. [12] Data is stored as nodes (entities) which are linked together by edges (relationships). Both nodes and edges can have properties (attributes) that describe the entity or relationship.
Nodes are typically described by nouns like persons, places, and products. Nodes can have different types in a graph database, making it possible to have both person and location nodes in the same graph, connecting them through edges.
Edges describe how the nodes are connected and how they relate to one another by linking them to each other.

As graph databases allow for easy retrieval of knowledge about relationships between nodes [13] we can understand that these type of databases are an excellent fit to store and represent data when the relationships between the data are of high importance.
In this project, the relationships between the different types of content will be a crucial part of the recommendation process to highlight potential patterns for decision-making.

2.5 Evaluating the recommendations

A recommender system's purpose is to suggest relevant content to users, so measuring and evaluating the results is a fundamental part of the development to keep a high standard showing the quality of the recommendation algorithm and assess its performance. Accordingly, there must be a measurable way to determine the system's accuracy. However, it is not without effort.

Evaluating how "good" a recommender system is will be a challenge when it's about whether a user considers a recommendation to be good or not. For us to be able to use standard metrics such as accuracy, we need to be able to classify an answer as either correct or incorrect. [14] This can be quite challenging when user feedback is subjective and influenced by a person's feelings, tastes and opinions. One way to use this subjective feedback is to translate it to objective feedback. This could be accomplished by asking the user leading follow-up questions that will make the conversation move in such a way so it results in objective observations or reactions.

However, while asking users for feedback might be the most accurate evaluation metric, even if we accomplish the task of translating user’s comments to objective reviews - the outcome is still hard to turn to a quantitative result.

Another way to evaluate the accuracy would be to measure the fraction of recommended articles out of total possible article recommendations, i.e. how many articles were recommended by the system for a given subject in contrast to the total amount of articles that are stored in the database.

3 Method

In this chapter, we will take a look at an example use case for the recommender system. After that, we will illustrate the system implementation, and lastly, section 3.3 explains how results will be evaluated.
This chapter contains shorter introductions to the technologies that will be used when building the system.

3.1 Problem scenario

In this project, our solution is supposed to provide support to an existing chatbot application built with .NET Core. The application's chatbot engine is using Dialogflow's [2] human-computer interaction service for building text-based conversational interfaces.

An everyday use case for the chatbot engine could be when a user wants to find information about a specific service that the website is offering. Figure 3.1 will show an example of such a conversation between a user and a chatbot agent.

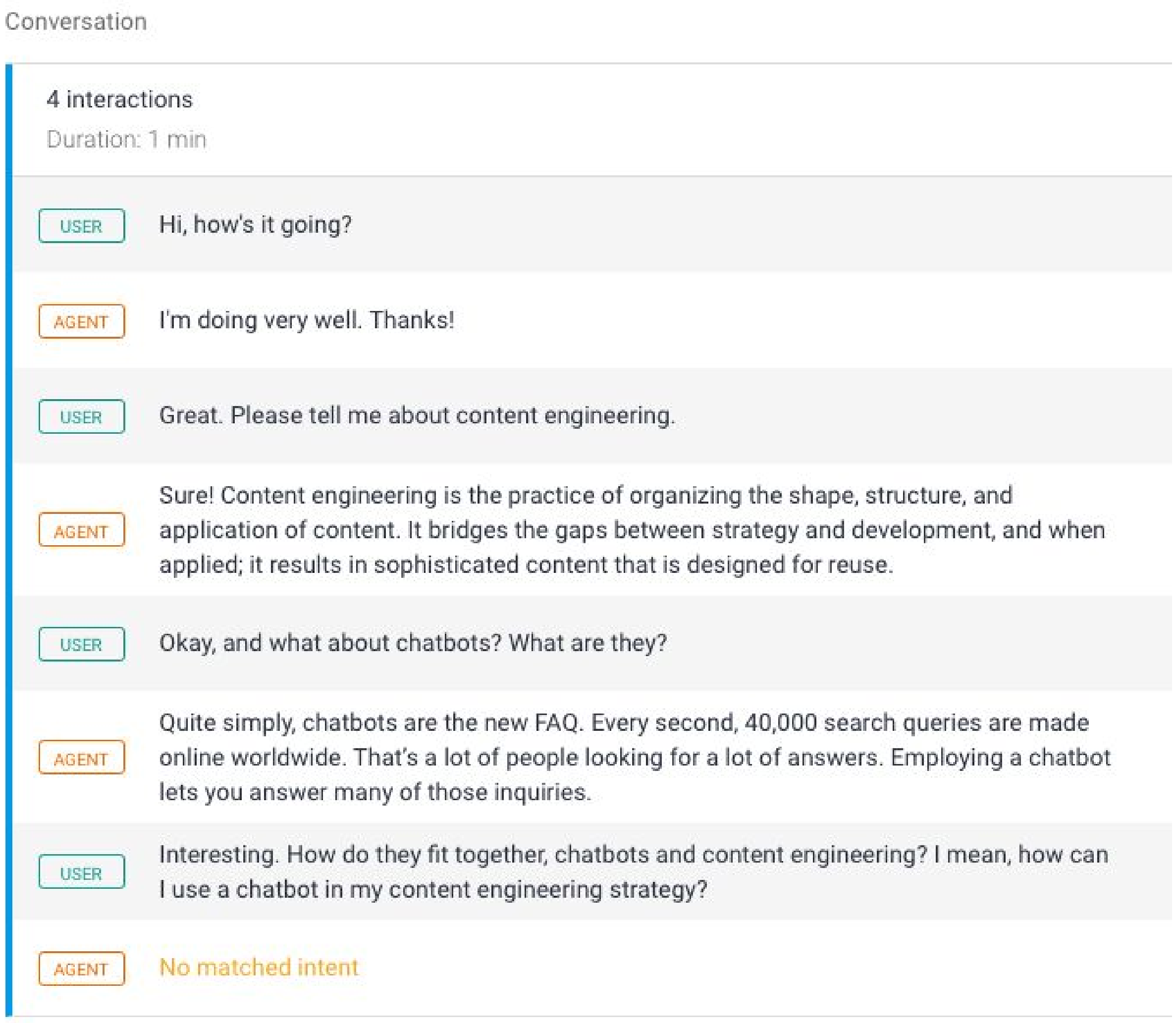
In this chapter, we will also take a look at how Dialogflow's service relates to the information presented in the overview of chatbot technology given in chapter 2.1.

It should be noticed that this project is basing its development on Dialogflow V1 which will be deprecated by the end of 2019. If this project work would be replicated after that time period, migrations to the latest version of Dialogflow might be needed.

3.1.1 Use case

The use case defined here is a situation when the user wants to find out information about a specific service that a website is offering, articles from [A]'s website [7] is being used as an example.

In this situation, the chatbot agent has been trained to know about the separate intents "content engineering" and "chatbots", but when the user asks a more advanced question of how these intents fit together - the chatbot agent will not be able to map the user's input to any predefined answer, as shown in figure 3.1. The chatbot agent will reply with a generic fallback intent, and it is in these use cases that the recommender system will be used.



Conversation

4 interactions
Duration: 1 min

USER Hi, how's it going?

AGENT I'm doing very well. Thanks!

USER Great. Please tell me about content engineering.

AGENT Sure! Content engineering is the practice of organizing the shape, structure, and application of content. It bridges the gaps between strategy and development, and when applied; it results in sophisticated content that is designed for reuse.

USER Okay, and what about chatbots? What are they?

AGENT Quite simply, chatbots are the new FAQ. Every second, 40,000 search queries are made online worldwide. That's a lot of people looking for a lot of answers. Employing a chatbot lets you answer many of those inquiries.

USER Interesting. How do they fit together, chatbots and content engineering? I mean, how can I use a chatbot in my content engineering strategy?

AGENT No matched intent

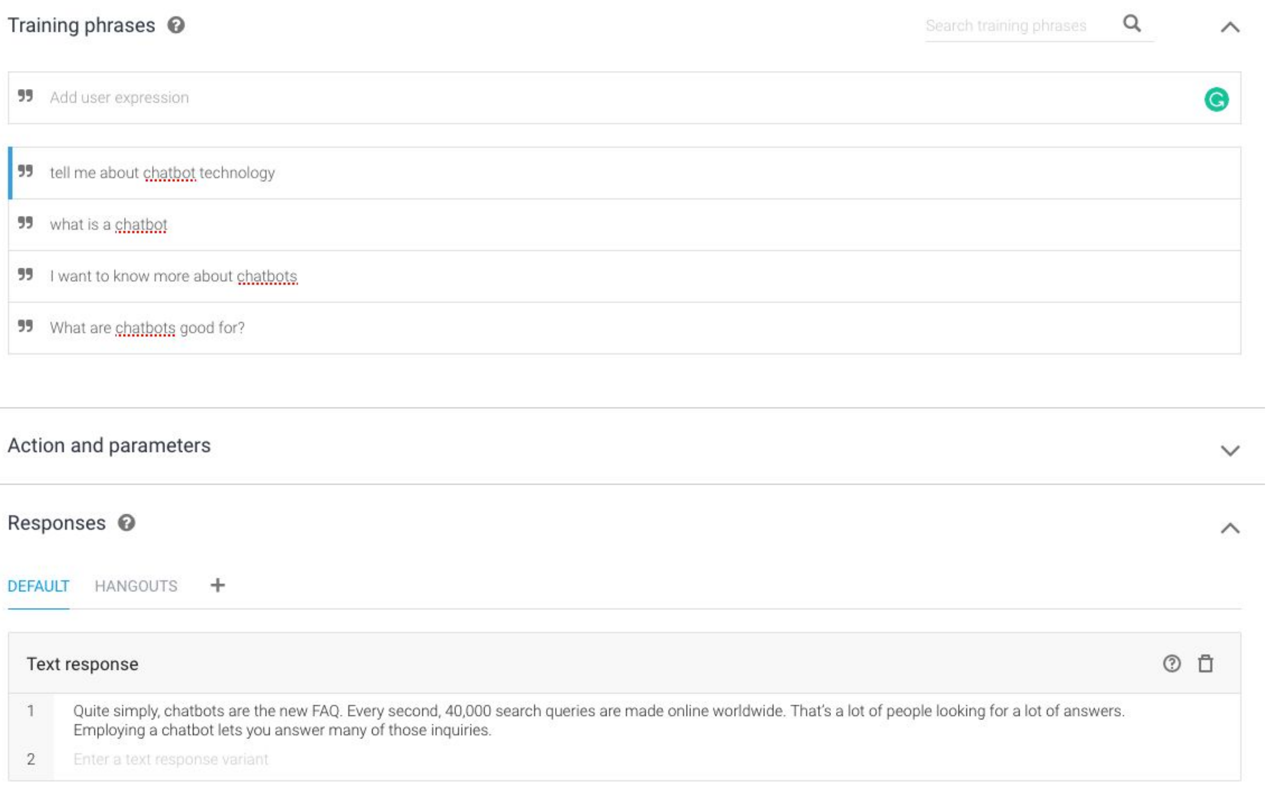
Figure 3.1: Demonstrating the chat between a user and a chatbot agent where the agent can't match the user's input to any redefined intent.

Let's take a brief look at how Dialogflow handles these conversations.

3.1.2 Dialogflow

Chatbot agents in Dialogflow are described as NLU modules. These agents in Dialogflow can be included in an application to transform user requests into actionable data.

Dialogflow handles conversation by allowing for creating intents. In each intent, examples of user utterances that can trigger the intent are defined as well as how the agent should respond. This way, an agent can map user input to a response. The agent's response usually prompts users for another utterance, which the agent will attempt to match to another intent, and the conversation continues. An example of how to create intents can be seen in figure 3.2.


Training phrases

Search training phrases

Add user expression

tell me about chatbot technology

what is a chatbot

I want to know more about chatbots

What are chatbots good for?

Action and parameters

Responses

DEFAULT HANGOUTS +

Text response

1 Quite simply, chatbots are the new FAQ. Every second, 40,000 search queries are made online worldwide. That's a lot of people looking for a lot of answers. Employing a chatbot lets you answer many of those inquiries.

2 Enter a text response variant

Figure 3.2: Showing how to create intents by providing training phrases representing user utterances and predefined responses for the agent to answer with.

To be able to know which criteria the recommender system is supposed to base its recommendations on, the chatbot agent needs to extract relevant data from the user's utterance and send this data to the system. In Dialogflow, a fulfillment service can be used for this purpose. [2, Fulfillment overview]

In our use case, we need a fulfillment that can query the recommender system for recommendations while providing the given subject(s) the user wants to know more about. In our example, these subjects would be "chatbots" and "content engineering".

3.2 System implementation

The recommender system design will consist of four components:

1. Web spider (crawler and scraper)
2. Bag-of-words model
3. Graph database
4. GraphQL API

The figure below shows an illustration of how the different components will interact with each other. The remaining subchapters will introduce the various parts, one by one.

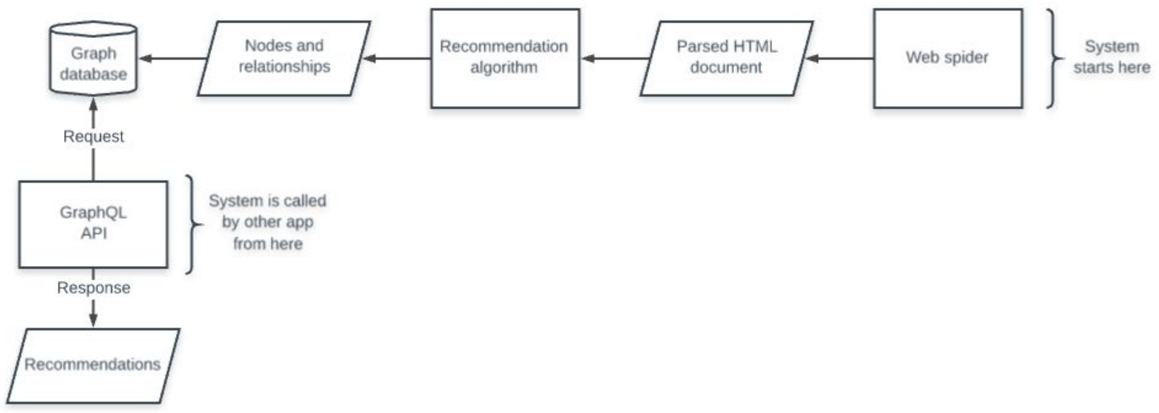


Figure 3.4: Showing how the different system components interact with each other and what data they produce as outcome.

3.2.1 DotnetSpider: web crawling & scraping library

This component will be implemented using the open-source library DotnetSpider [15], a .NET standard web crawling and scraping library. It is installed via NuGet package manager [16] and supports .NET Core.

The web spider component will need several demands to be met, they are:

- **A queue system for links:** it's impossible for a fast web crawler to handle all links in memory. The best approach is to create a list or queue, that you push links onto for crawling. [10] As the spider moves from one web page to the next, it needs to pick up a new link to crawl. That new link can be pulled from the queue. As the spider discovers new links, they can be pushed to the queue, waiting for discovery by the next crawl process.
- **A spider class:** that allows for custom implementations such as change of crawling speed, a restriction for which URLs the spider will follow, as well as providing opportunities for inheritance where subclasses can inherit from the spider class and override its methods with their implementations.
- **A data parser class:** DotnetSpider library uses XPath which is a language that locates and parses specific information from a web page's source code. [17] XPath is used to navigate HTML documents - either by searching for tag names, classes and other tag attributes, or through specific positions in the document. As mentioned in chapter 2.3.2, because of the lack of common practice on how element markings are used there is still a need to examine a website's structure before concluding on how to extract the requested

information from the page. DotnetSpider library provides opportunities for inheritance where subclasses can inherit from the data parser class and override its methods with their implementations.

- **Data storage options:** default storage options that come with DotnetSpider library ranges from console storage to database integrations and the creation of JSON-files. It also offers the opportunity to use a BaseStorage interface when creating your custom storage options.
- **A scheduler:** this function can be used for two reasons; either to schedule regular crawlings for websites that are updated frequently with new content, or for sites that need to consider the fact that servers can be under pressure at certain times of a day. Scheduling the crawling procedure is an intelligent approach to ease that pressure on the server.

DotnetSpider library was chosen because of its support for the features required in our web spider component. An illustration of the architecture of the DotnetSpider library can be seen in figure 3.5.

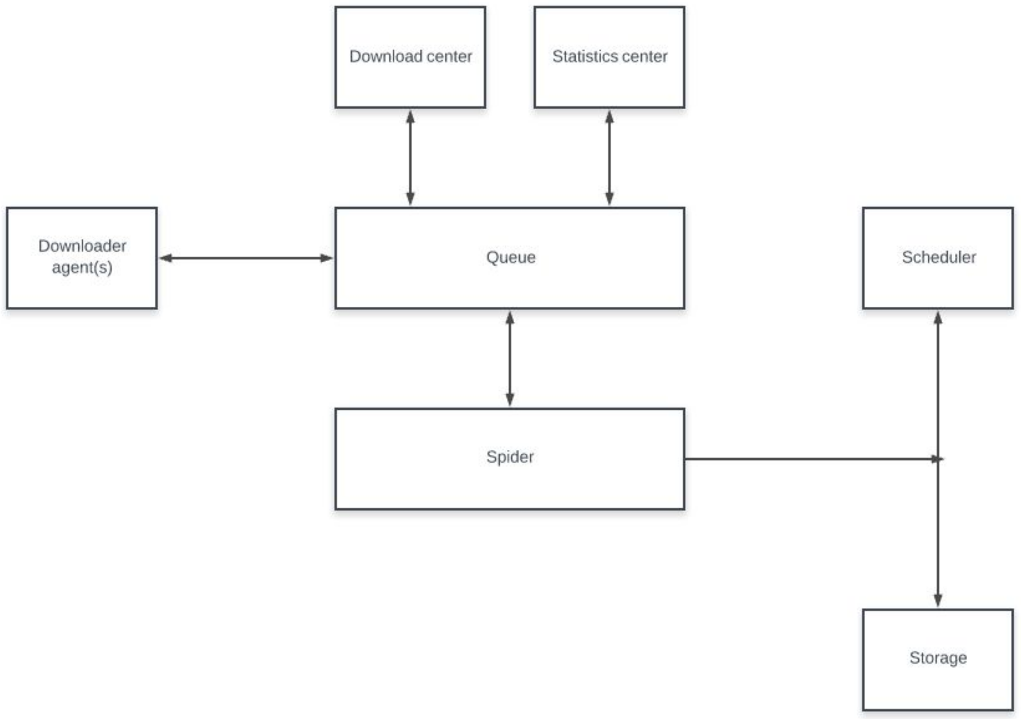


Figure 3.5: Illustrating the architecture of DotnetSpider library.

The web spider's process of retrieving, parsing and saving information is described below and illustrated in figure 3.6.

1. When the user chooses to start the recommender system application, a specified start URL will be added to the queue.
2. The program checks whether the URL link is valid and if it is; it's visited.

3. The HTML source code of the web page is downloaded.
4. The source code is parsed to find specified elements.
5. The information found is stored with the chosen storage technique; in our example, we will use a JSON file as storage.
6. The link will be removed from the queue and added to a list containing already visited links.
7. This process will continue until the queue no longer contains any links to visit.

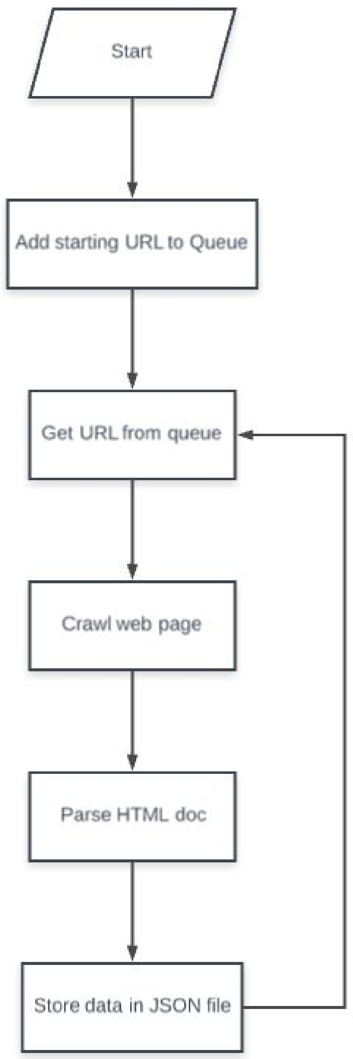


Figure 3.6: A flowchart showing the process of the web spider component.

3.2.2 Bag-of-Words model

The bag-of-words model component will continue where the web spider left off. The content of each webpage is now stored in a JSON-file, which this component will get a hold of and use. In the JSON-file, each line will represent a JSON object, and that object will represent an article. The article object has information about its URL, the title, meta keywords, meta description and all paragraphs. This is illustrated in figure 3.7.

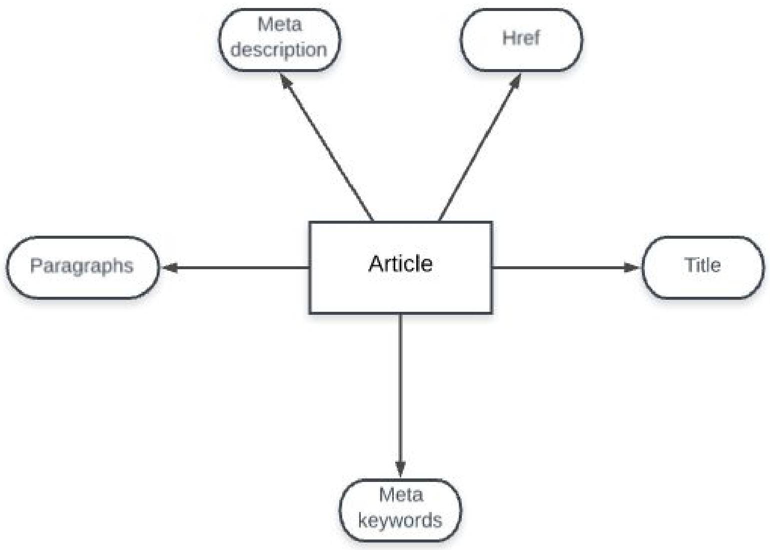


Figure 3.7: An illustration of how an article object is stored in the JSON-file.

This component has to review and calculate a numeric scoring for each content item on a web page to evaluate how likely it is for that page to be an appropriate recommendation on a particular subject.
The bag-of-words model needs to go through a sequence of steps to get that score, and this process is described below:

1. Pre-process the text.
2. Make a list of all unique words.
3. Apply a numeric score to each word in the text.

Regarding the text pre-processing, this step helps to avoid problems that are often encountered in data processing like missing values and irrelevant data - both common issues when dealing with most real-world data sources. [18]
In our case, the data we want to use is all text data, so the component will need to perform transformations on the text to prepare it for the next steps. This is called text normalization and is the process of transforming the text into a single canonical form. When you normalize a text, you remove punctuation, remove trailing whitespaces and convert the text to all lowercase, among other transformations.

When the data processing is ready, it’s time to tokenize the text into words. Tokenizing the text is when you break the text apart into single words, ending up with a list of all unique words. Here we filter out stop words. Stop words usually refers to the most common words in a language or frequent words that don't contain much information [18]; like "a", "the", "that", etc.

At this point, we will have a vocabulary of single words that appear in the text, excluding stop words. Once we've reached here, the occurrence of each word needs to be scored. The simplest scoring method is to mark the presence or absence of words in the text, but as we want to compare all articles against each other to find the highest scores (and presumably the most relevant recommendation), some additional scoring methods are necessary. These are:

- **Count.** We will count the number of times each word appears in a text.
- **Frequency**. We will calculate the density that each word appears in a text out of all the words in that text.

The BoW model was chosen because it's simple to understand and implement and has seen great success in problems such as document classification. [18]
After this component has finished, we will have article and paragraph objects with numeric scorings for word count and word frequency.

Next, it is time for the graph database to take over the responsibility of the data.

3.2.3 Neo4j graph database

The graph database component will be built with Neo4j graph platform. [19] Neo4j is a native graph processing engine that supports high-performance graph queries on large datasets. One thing that is unique about graph databases is the data model; graph databases focus on the relationship between the data, so Neo4j is optimized for working with complex and connected data.

Neo4j was chosen because of the need to traverse the continually growing, highly linked dataset. The web spider component will add new data or update changes to existing data each time a new crawling session has found information to renew. This means the dataset will be continually growing. It is also highly linked because all the different content parts belonging to the same web page, and all web pages come from the same website.

When adding information to Neo4j database, the following data structure will be used:

- Article nodes with attributes of url, title, keywords (meta keywords) and summary (meta description). One article node will represent one web page.

- Paragraph nodes with attributes of the article’s url and the text from the paragraph. One paragraph node will represent one paragraph in an article.
- Word nodes. One word node will represent one word from the paragraph text.

This structure and all the relationships that connect the nodes are illustrated in figure 3.8.

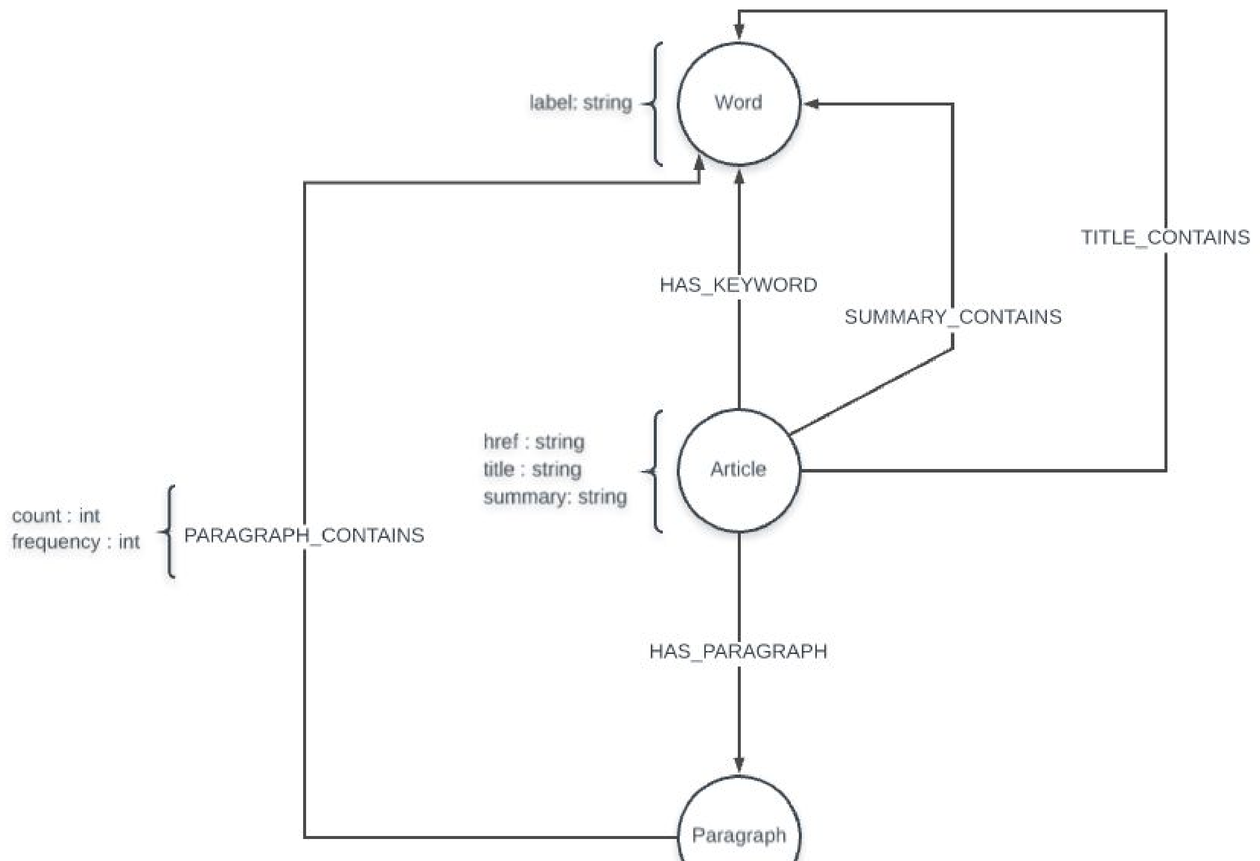


Figure 3.8: Illustration of how nodes and edges will be structured in the graph database.

The querying language that will be used for interacting with Neo4j will be Cypher Graph Query Language [19, Cypher Graph Query Language]. Cypher is inspired by SQL’s concept of pattern matching, but designed for graphs instead of tables.
A simple example comparison between Cypher and SQL can be seen in figure 3.9.

```
MATCH (a:Article)-[:TITLE_CONTAINS]->(w:Word { Label: “chatbots” })
RETURN a, w

SELECT * FROM Article
WHERE Title LIKE “chatbots”;
```

Figure 3.9: Illustrating a comparison between Cypher Graph Query Language and SQL.

During the development of the recommender system, we will take advantage of the Neo4j browser [19, The Neo4j Browser: A User Interface Guide for Beginners] which is a tool for interacting with Neo4j. This tool makes it easy to visualize our stored data in a graph format.

As the goal with this graph database component is to be a part of the complete recommender system, C# code will be used to query the database. Neo4j has language drivers for the most popular programming languages that allow for sending cypher queries to the database.
For this project, Neo4j Bolt driver for .NET [20] will be used in the system application, which is the official Neo4j .NET driver for connecting to Neo4j 3.0.0+ databases.
The Neo4j driver is distributed exclusively through NuGet Package Manager.

3.2.4 GraphQL queries

The last component is the GraphQL API. GraphQL [21] is a specification for querying an application graph. The queries have a tree structure based on entities and relevant attributes and any of those can take parameters for filtering and pagination.

GraphQL provides the opportunity to read queries (retrieve information), make mutations (create, update or delete information), and subscribe for live updates when some event happens.

For this component, GraphQL for .NET [22] will be used as it’s an implementation of GraphQL in .NET that can be installed via NuGet Package Manager.

3.3 System evaluation

The requirements and evaluation methods are needed to ensure that the system is useful from a functional point of view. Each component should be tested individually to see that they work as expected, as well as how they work together to produce the outcome.

The system evaluation will be based on the purpose of the project, which is to support a chatbot application’s fallback experience by using a content-based recommender system. Therefore, the foundation of the evaluation will be technical, where we confirm that all components of the system work as expected.

I will also do a limited evaluation of the recommended results. In chapter 2.5, some ways to evaluate a recommender system was introduced. As performing user testing is out of the scope in this project, the performance will be evaluated by studying the system’s ability to produce relevant recommendations in the mentioned use case scenario. We will consider if the recommended articles and paragraphs seems to, by a general view, be relevant to the user’s input.

4 Implementation

This chapter will begin by presenting the development environment and chosen project setup. Then it will continue with going through the implementation phase for each component of the recommender system.

4.1 Development environment and project setup

The IDE used during the building of the recommender system is Visual Studio. [23] Visual Studio makes it easy to get up and running with building .NET Core applications. The programming language that will be used is C#.

The current chatbot application solution consists of several projects, each having specific responsibilities and all are incorporated in the main solution. One project is responsible for the front-end of the chatbot, another project is responsible for the core functionality of the chatbot, and so forth.

To follow that same structure, I chose to set up the recommender system module as two separate .NET projects interacting with each other. Both have their distinct responsibilities and are joined with the rest of the main solution:

- A console app that handles the web crawling, scraping and data processing parts.
- A web API with .NET Core to handle the connection with the database.

4.2 Web spider

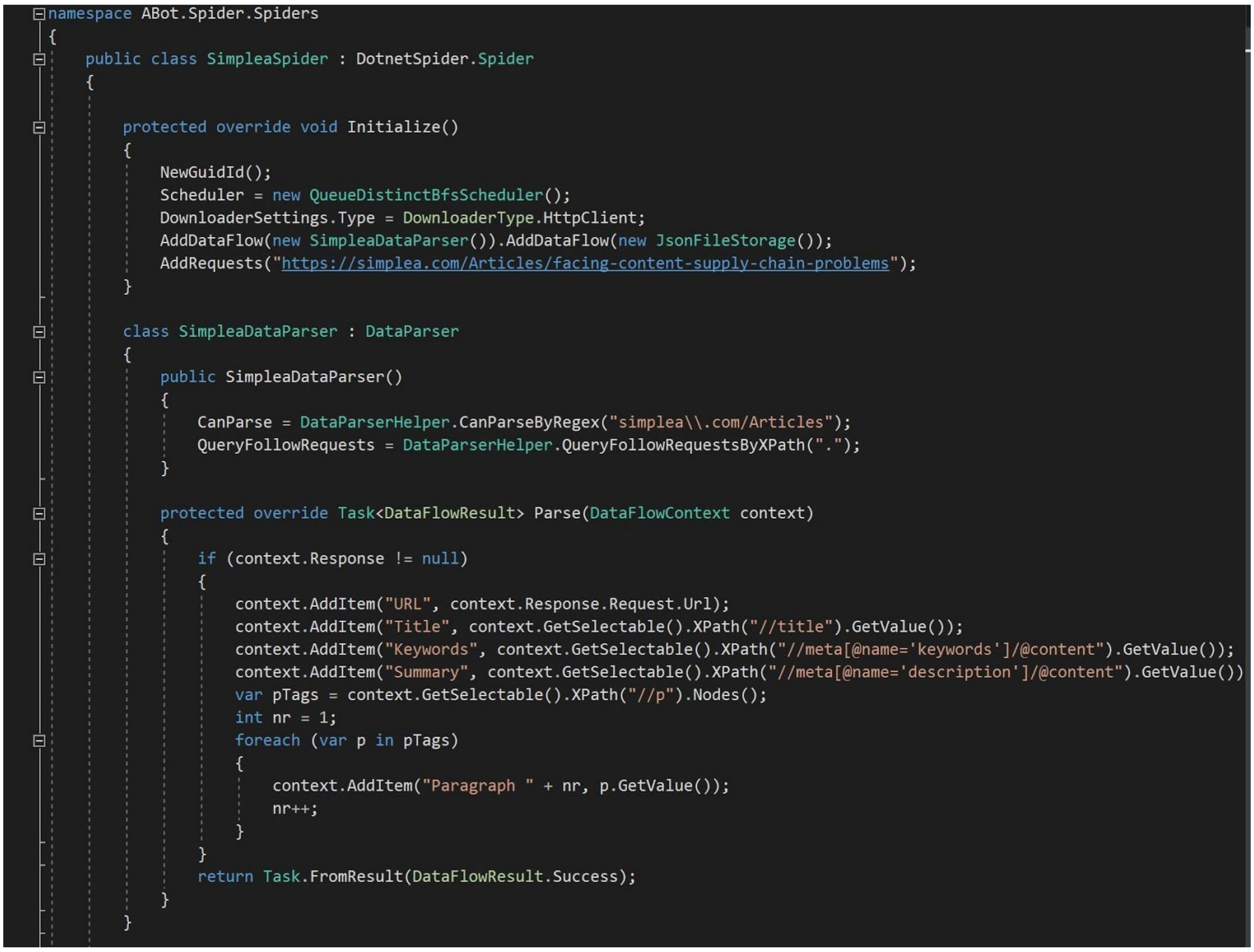
We start by building the web spider component. This component's primary responsibility will be to retrieve data from web pages. Without the crawler running as the first step of this system, the rest of its components will have no data to execute their responsibilities.

As mentioned in chapter 2.3.2, the semi-structured nature of HTML documents makes it difficult to extract information from a web page without knowing the structure beforehand. To solve this problem, we will use the opportunity provided by DotnetSpider library; namely, the ability to create subclass spiders for each website - and all of these subclasses will inherit from the superclass Spider.
As we're using [A]'s website in our example use case, we create a SimpleaSpider class. Its source code can be found in figure 4.1.
This class inherits from DotnetSpider.Spider, and its responsibilities are:

- To create a unique Id for each crawler.

- To add a data parser subclass and a storage type.
- To add a start URL as the first request to the link queue.
- To set the query that decides which links to follow.

As noticed in chapter 2.3.3, most spiders are implemented according to guidelines and policies. One policy that our spider will consider is the “selection policy”. Each spider is supposed to crawl web pages of one single website at a time. Therefore, the SimpleaSpider class will only add links to the queue if they belong to the same domain.



```
namespace ABot.Spider.Spiders
{
    public class SimpleaSpider : DotnetSpider.Spider
    {

        protected override void Initialize()
        {
            NewGuidId();
            Scheduler = new QueueDistinctBfsScheduler();
            DownloaderSettings.Type = DownloaderType.HttpClient;
            AddDataFlow(new SimpleaDataParser()).AddDataFlow(new JsonFileStorage());
            AddRequests("https://simplea.com/Articles/facing-content-supply-chain-problems");
        }

        class SimpleaDataParser : DataParser
        {
            public SimpleaDataParser()
            {
                CanParse = DataParserHelper.CanParseByRegex("simplea\\.com/Articles");
                QueryFollowRequests = DataParserHelper.QueryFollowRequestsByXPath(".");
            }

            protected override Task<DataFlowResult> Parse(DataFlowContext context)
            {
                if (context.Response != null)
                {
                    context.AddItem("URL", context.Response.Request.Url);
                    context.AddItem("Title", context.GetSelectable().XPath("//title").GetValue());
                    context.AddItem("Keywords", context.GetSelectable().XPath("//meta[@name='keywords']/@content").GetValue());
                    context.AddItem("Summary", context.GetSelectable().XPath("//meta[@name='description']/@content").GetValue())
                    var pTags = context.GetSelectable().XPath("//p").Nodes();
                    int nr = 1;
                    foreach (var p in pTags)
                    {
                        context.AddItem("Paragraph " + nr, p.GetValue());
                        nr++;
                    }
                }
                return Task.FromResult(DataFlowResult.Success);
            }
        }
```

Figure 4.1: The source code of SimpleaSpider class

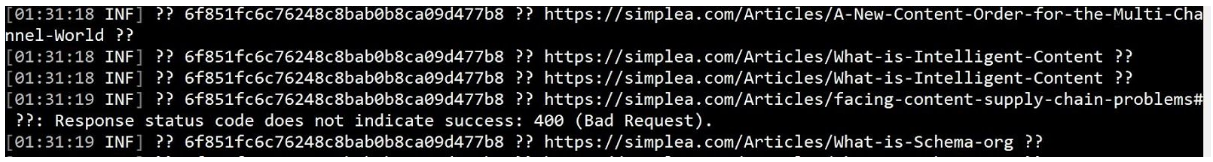
As seen in the source code, the content retrieved by the scraper is the URL of the web page, its title, the meta keywords, the meta description, and the text paragraphs.
In the console application’s Main method, a spider will be initiated and it’s Run() method will be called. The Run() method is a synchronous call but it also has an

asynchronous counterpart called RunAsync(). We do not need to use asynchronous programming at this stage in the development, as no multiple processes are running simultaneously in parallel. When the user wants to crawl a website, he/she can start the web crawling console application.

```
var spider = Spider.Create<SimpleaSpider>();
spider.Run();
```

Figure 4.2: Showing how to create a spider in the Main method.

When starting the web spider component, some information about the crawling process will be displayed in the console for each crawled web page. That information is the spider id and the crawled URL. If the server sent a bad request response to the spider that will also be visible in the console.

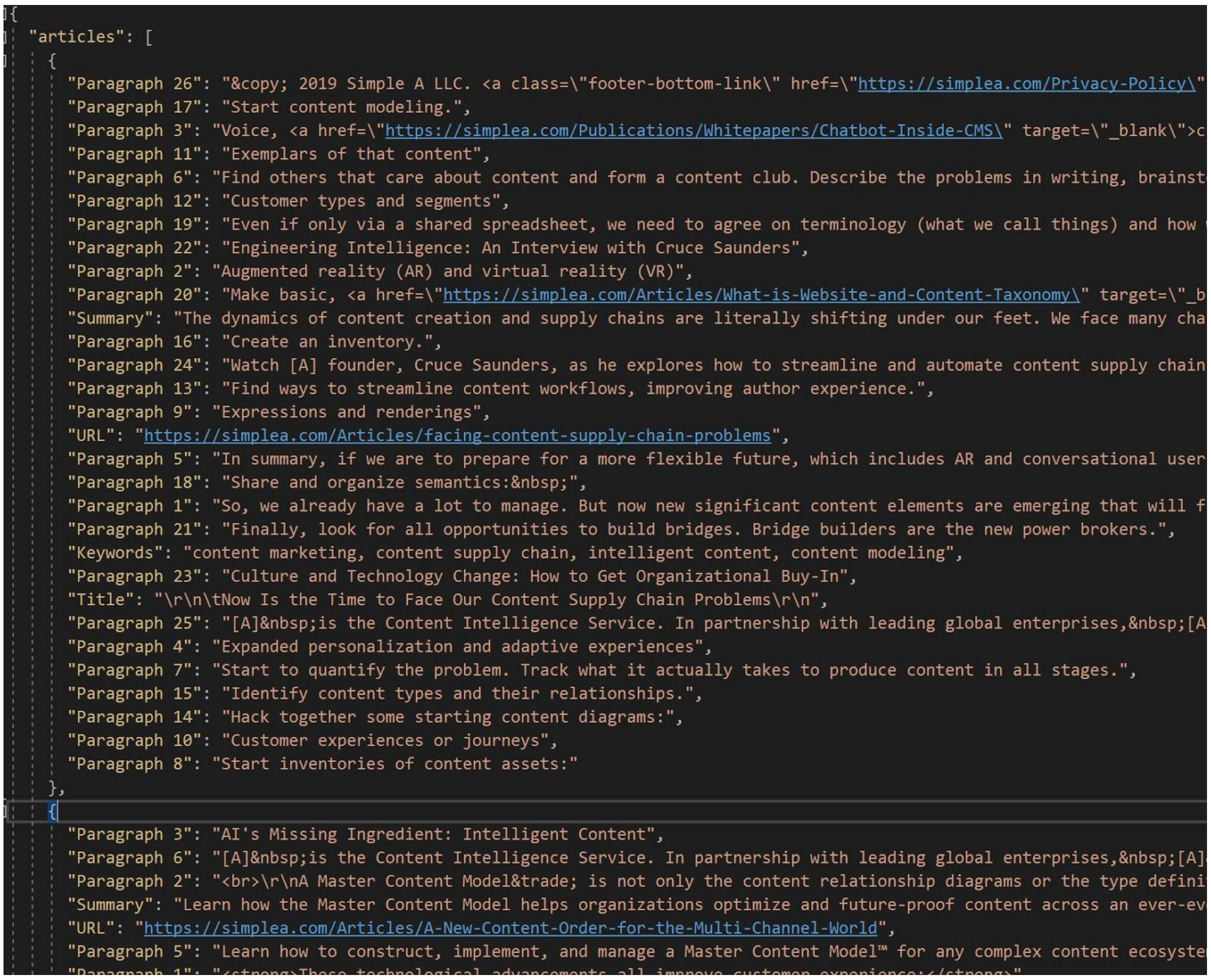


```
[01:31:18 INF] ?? 6f851fc6c76248c8bab0b8ca09d477b8 ?? https://simplea.com/Articles/A-New-Content-Order-for-the-Multi-Cha
nnel-World ??
[01:31:18 INF] ?? 6f851fc6c76248c8bab0b8ca09d477b8 ?? https://simplea.com/Articles/What-is-Intelligent-Content ??
[01:31:18 INF] ?? 6f851fc6c76248c8bab0b8ca09d477b8 ?? https://simplea.com/Articles/What-is-Intelligent-Content ??
[01:31:19 INF] ?? 6f851fc6c76248c8bab0b8ca09d477b8 ?? https://simplea.com/Articles/facing-content-supply-chain-problems#
 ??: Response status code does not indicate success: 400 (Bad Request).
[01:31:19 INF] ?? 6f851fc6c76248c8bab0b8ca09d477b8 ?? https://simplea.com/Articles/What-is-Schema-org ??
```

Figure 4.3: An example of the console output while the web spider is crawling a website.

The spider continues with its process until it has gone through all URLs in the queue. An empty URL queue might mean that the spider has gone through all web pages it has encountered on the website. But, it might also mean that the server has some sort of mechanism installed that blocks a spider after X amount of requests, making the remaining URLs in the queue returning only bad requests and therefore, no more URLs will be added to the queue.

When the crawling has finished, a JSON-file is created which contains the parsed data, see figure 4.4 for an example for such a file.
The name of this file is the id of the spider that just performed the crawling session.



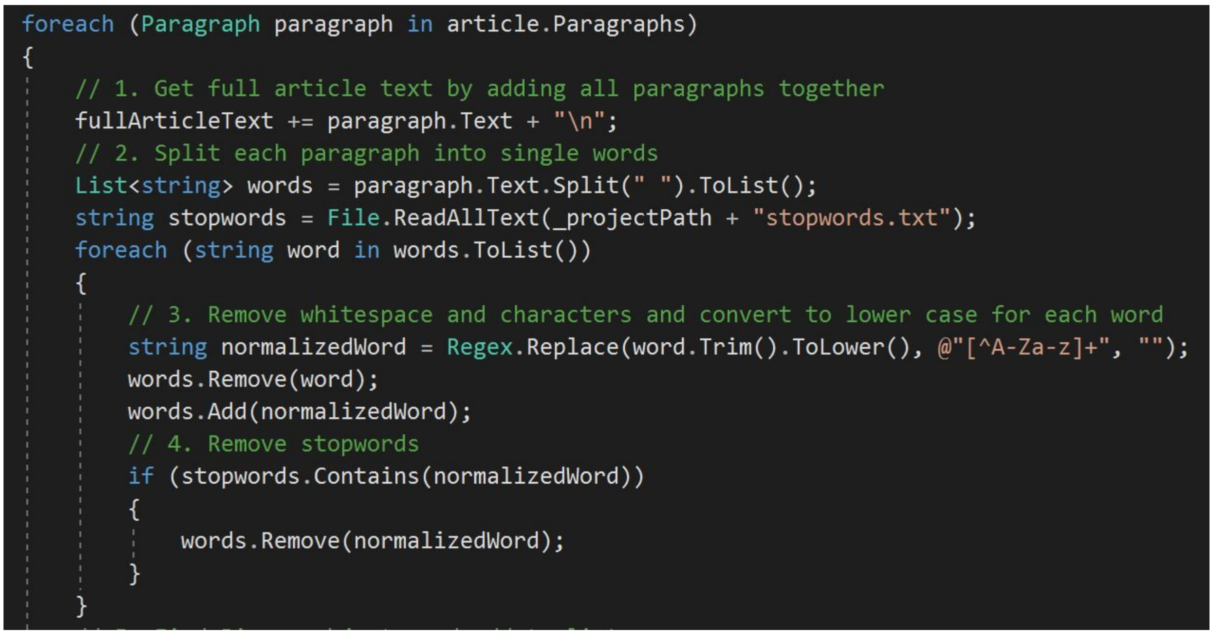
```json
{
  "articles": [
    {
      "Paragraph 26": "&copy; 2019 Simple A LLC. <a class=\"footer-bottom-link\" href=\"https://simplea.com/Privacy-Policy\"
      "Paragraph 17": "Start content modeling.",
      "Paragraph 3": "Voice, <a href=\"https://simplea.com/Publications/Whitepapers/Chatbot-Inside-CMS\" target=\"_blank\">c
      "Paragraph 11": "Exemplars of that content",
      "Paragraph 6": "Find others that care about content and form a content club. Describe the problems in writing, brainst
      "Paragraph 12": "Customer types and segments",
      "Paragraph 19": "Even if only via a shared spreadsheet, we need to agree on terminology (what we call things) and how
      "Paragraph 22": "Engineering Intelligence: An Interview with Cruce Saunders",
      "Paragraph 2": "Augmented reality (AR) and virtual reality (VR)",
      "Paragraph 20": "Make basic, <a href=\"https://simplea.com/Articles/What-is-Website-and-Content-Taxonomy\" target=\"_b
      "Summary": "The dynamics of content creation and supply chains are literally shifting under our feet. We face many cha
      "Paragraph 16": "Create an inventory.",
      "Paragraph 24": "Watch [A] founder, Cruce Saunders, as he explores how to streamline and automate content supply chain
      "Paragraph 13": "Find ways to streamline content workflows, improving author experience.",
      "Paragraph 9": "Expressions and renderings",
      "URL": "https://simplea.com/Articles/facing-content-supply-chain-problems",
      "Paragraph 5": "In summary, if we are to prepare for a more flexible future, which includes AR and conversational user
      "Paragraph 18": "Share and organize semantics:&nbsp;",
      "Paragraph 1": "So, we already have a lot to manage. But now new significant content elements are emerging that will f
      "Paragraph 21": "Finally, look for all opportunities to build bridges. Bridge builders are the new power brokers.",
      "Keywords": "content marketing, content supply chain, intelligent content, content modeling",
      "Paragraph 23": "Culture and Technology Change: How to Get Organizational Buy-In",
      "Title": "\r\n\tNow Is the Time to Face Our Content Supply Chain Problems\r\n",
      "Paragraph 25": "[A]&nbsp;is the Content Intelligence Service. In partnership with leading global enterprises,&nbsp;[A
      "Paragraph 4": "Expanded personalization and adaptive experiences",
      "Paragraph 7": "Start to quantify the problem. Track what it actually takes to produce content in all stages.",
      "Paragraph 15": "Identify content types and their relationships.",
      "Paragraph 14": "Hack together some starting content diagrams:",
      "Paragraph 10": "Customer experiences or journeys",
      "Paragraph 8": "Start inventories of content assets:"
    },
    {
      "Paragraph 3": "AI's Missing Ingredient: Intelligent Content",
      "Paragraph 6": "[A]&nbsp;is the Content Intelligence Service. In partnership with leading global enterprises,&nbsp;[A]
      "Paragraph 2": "<br>\r\nA Master Content Model&trade; is not only the content relationship diagrams or the type defini
      "Summary": "Learn how the Master Content Model helps organizations optimize and future-proof content across an ever-ev
      "URL": "https://simplea.com/Articles/A-New-Content-Order-for-the-Multi-Channel-World",
      "Paragraph 5": "Learn how to construct, implement, and manage a Master Content Model™ for any complex content ecosyste
      "Paragraph 1": "<strong>These technological advancements all improve customer experience:</strong>"
```

Figure 4.4: An example of a JSON-file that the web spider uses to store the parsed data.

4.3 Bag-of-Words model

This component is represented in the application by a class named DataProcessor. When this class receives the JSON data, it will start the process mentioned in chapter 3.2.2. The text needs to be normalized and stop words will be removed before we can apply the scoring. As there is no single universal list of stop words, we need to make one for ourselves and include that in the project. This will be a normal text file containing each stop word on a separate line, named **stopwords.txt**.

Figure 4.5 demonstrates the source code of this process. The class will go through each paragraph from the web page, split the text into single words, normalize the words by removing whitespace and punctuation, and lastly removing all stop words.



```
foreach (Paragraph paragraph in article.Paragraphs)
{
    // 1. Get full article text by adding all paragraphs together
    fullArticleText += paragraph.Text + "\n";
    // 2. Split each paragraph into single words
    List<string> words = paragraph.Text.Split(" ").ToList();
    string stopwords = File.ReadAllText(_projectPath + "stopwords.txt");
    foreach (string word in words.ToList())
    {
        // 3. Remove whitespace and characters and convert to lower case for each word
        string normalizedWord = Regex.Replace(word.Trim().ToLower(), @"[^A-Za-z]+", "");
        words.Remove(word);
        words.Add(normalizedWord);
        // 4. Remove stopwords
        if (stopwords.Contains(normalizedWord))
        {
            words.Remove(normalizedWord);
        }
    }
```

Figure 4.5: The process of normalization and stop words removal

The last step would be to get the word count.
Let’s take this paragraph as an example:
“In order to give chatbots the content they need, we need to understand how chatbots work.”

After pre-processing the data, the remaining words will be divided into a list of single words, like the following:

“order”,
“give”,
“chatbots”,
“content”,
“understand”,
“chatbots”,
“work”

As you might have noticed, the word “chatbots” appears twice in this paragraph and we need to represent that as a numeric score.
We could take advantage of a C# Dictionary here, having the single words as keys and scores as values. We calculate how many times each single word is repeated in the list so that instead of having repeated words, we add one to the numeric score of that word.
An example of such a Dictionary is shown below:

“order”, 1

"give", 1
"chatbots", 2
"content", 1
"understand", 1
"work", 1

The last step of the data processing would be to calculate the word frequency. Here we would measure the density of the word.
When calculating word frequency we need to know how many words the paragraph contains in total. In our example, that is 16 words which means that the estimation would be:

2 / 16
word count / total word amount

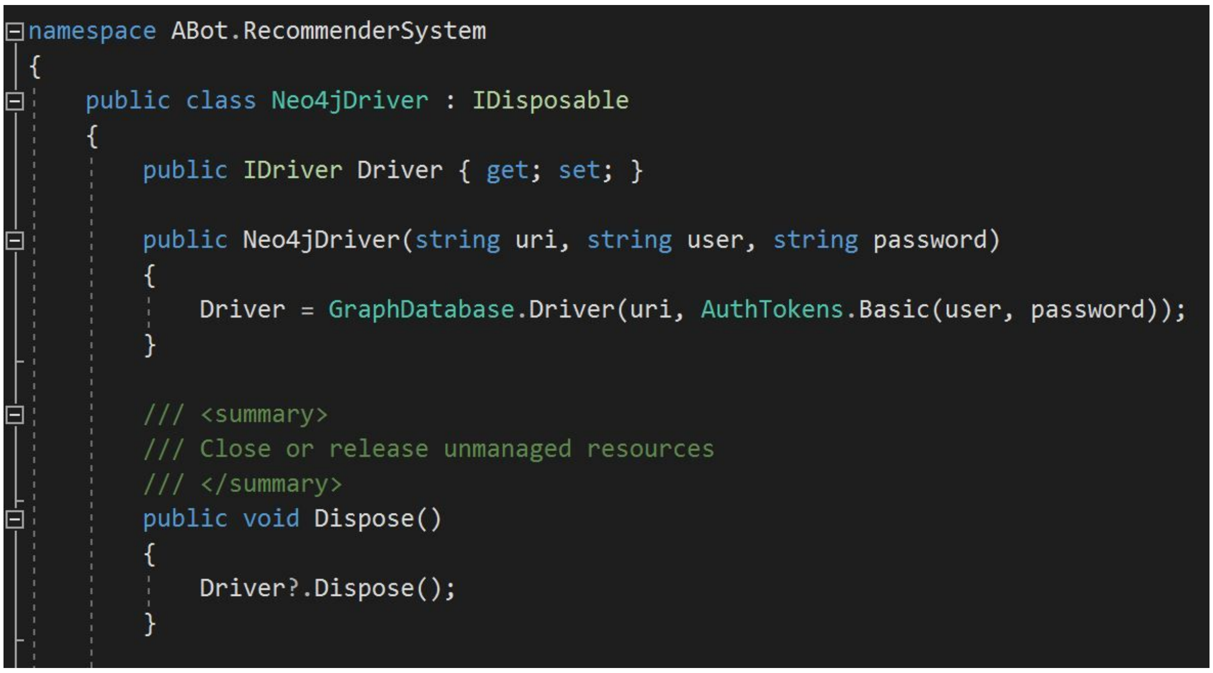
This will leave a frequency rate at just below 13%. So, in this paragraph used as an example - the word "chatbots" will have a count of 2, and a frequency rate of 13%.
As we also need to store the frequency in our Dictionary, we'll make a slight change to its structure; we still use the word as the key and use an array of integers as the value. The accurate frequency rate in this case was 12,5%, but we can round off the number to the highest or lowest integer using Math.Round() - making it slightly easier for us to handle the values in this basic application.
After this update, the Dictionary would look like the following:
"order", [1, 6]
"give", [1, 6]
"chatbots", [2, 13]
"content", [1, 6]
"understand", [1, 6]
"work", [1, 6]

The scores will be stored in the graph database as properties to the relationship PARAGRAPH_CONTAINS that connects a paragraph node and a word node, as can be seen in figure 3.8.

4.4 Graph database

Up until now, we've been focusing on the console application project that handles the crawling, scraping and data processing. The console application runs in a linear way and when it has reached its end, it will send the information to the web API project for querying and storing the data in Neo4j database.

The web API project has a class called Neo4jDriver that handles the connection to the database. This class uses Neo4j Driver package for .NET. The basic setup for this driver can be seen in figure 4.6.



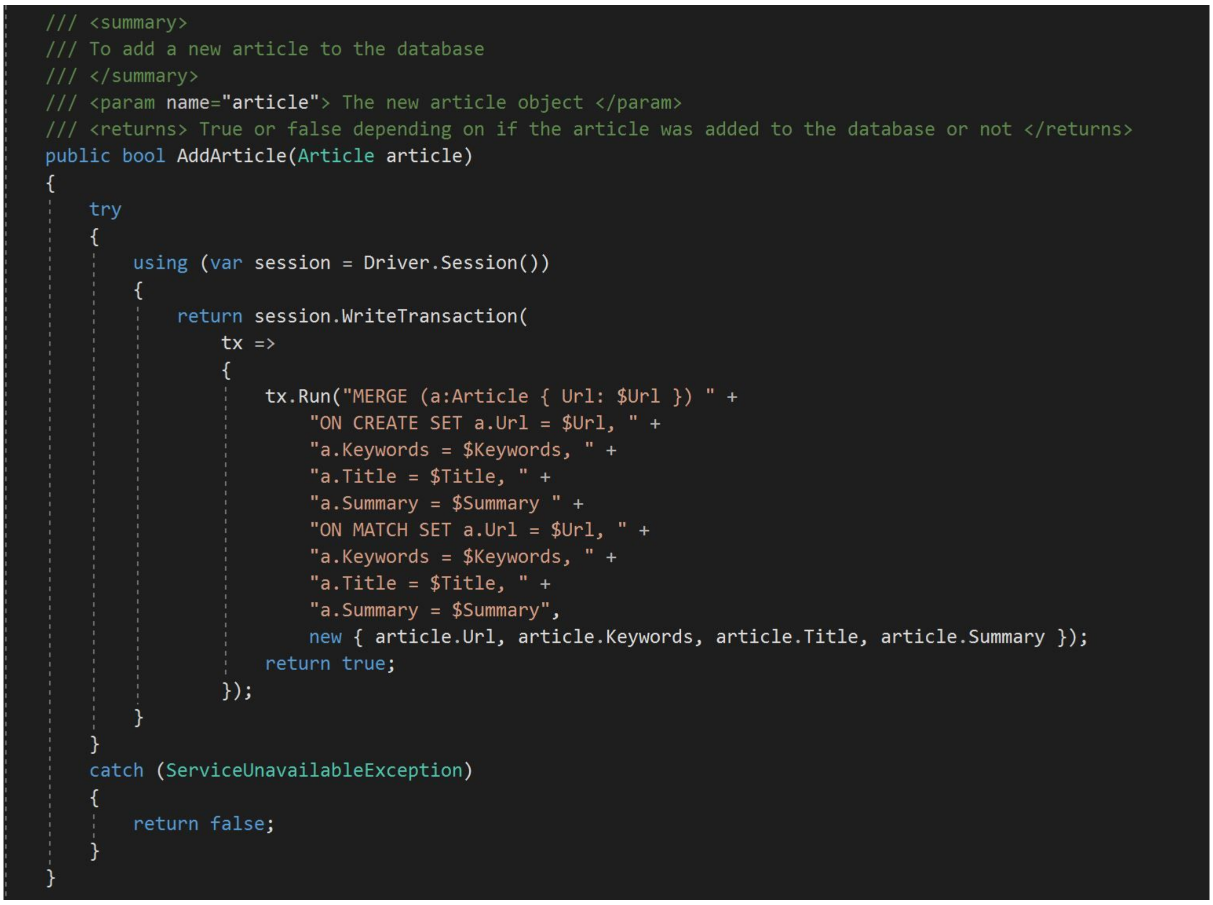
```
namespace ABot.RecommenderSystem
{
    public class Neo4jDriver : IDisposable
    {
        public IDriver Driver { get; set; }

        public Neo4jDriver(string uri, string user, string password)
        {
            Driver = GraphDatabase.Driver(uri, AuthTokens.Basic(user, password));
        }

        /// <summary>
        /// Close or release unmanaged resources
        /// </summary>
        public void Dispose()
        {
            Driver?.Dispose();
        }
```

Figure 4.6: Showing the basic setup of Neo4j Driver in .NET

Neo4jDriver class will have separate methods for adding article, paragraph and word nodes. The method needs to be provided with the data object representing the node so that it can retrieve its information to use as properties for the node.
An example of these methods can be seen in figure 4.7. Here it can be noticed that we use the MERGE command instead of CREATE in Cypher when querying the database. That's because we want to avoid duplicates of nodes so if this article already exists, its information will be updated - otherwise a new node will be created. Therefore, this method can be used both when creating new articles or when updating already existing articles whose content might have changed in some way.



```
/// <summary>
/// To add a new article to the database
/// </summary>
/// <param name="article"> The new article object </param>
/// <returns> True or false depending on if the article was added to the database or not </returns>
public bool AddArticle(Article article)
{
    try
    {
        using (var session = Driver.Session())
        {
            return session.WriteTransaction(
                tx =>
                {
                    tx.Run("MERGE (a:Article { Url: $Url }) " +
                        "ON CREATE SET a.Url = $Url, " +
                        "a.Keywords = $Keywords, " +
                        "a.Title = $Title, " +
                        "a.Summary = $Summary " +
                        "ON MATCH SET a.Url = $Url, " +
                        "a.Keywords = $Keywords, " +
                        "a.Title = $Title, " +
                        "a.Summary = $Summary",
                        new { article.Url, article.Keywords, article.Title, article.Summary });
                    return true;
                });
        }
    }
    catch (ServiceUnavailableException)
    {
        return false;
    }
}
```

Figure 4.7: Adding an article to the database via Neo4jDriver class

We create paragraph and word nodes in the same way, making sure to also create relationships between the nodes as we're adding new data to the database.

After all data has been added, we can use Neo4j Browser to view the data in a graph format. In figure 4.8 we see the example of an article from [A]'s website with the url *https://simplea.com/Articles/AI-Marketing-Chatbots-and-Your-CMS*.

In the graph we can see the article node in orange, connected to many paragraph nodes in a light brown colour - associated with each other through the HAS_PARAGRAPH relationship.
We can also see that the article node connects to several word nodes in blue colour through relationships like TITLE_CONTAINS and SUMMARY_CONTAINS, etc.
Last but not least, we notice that paragraph nodes also connect to word nodes through the PARAGRAPH_CONTAINS relationship. It's this type of relationship that contains the properties of word count and frequency.

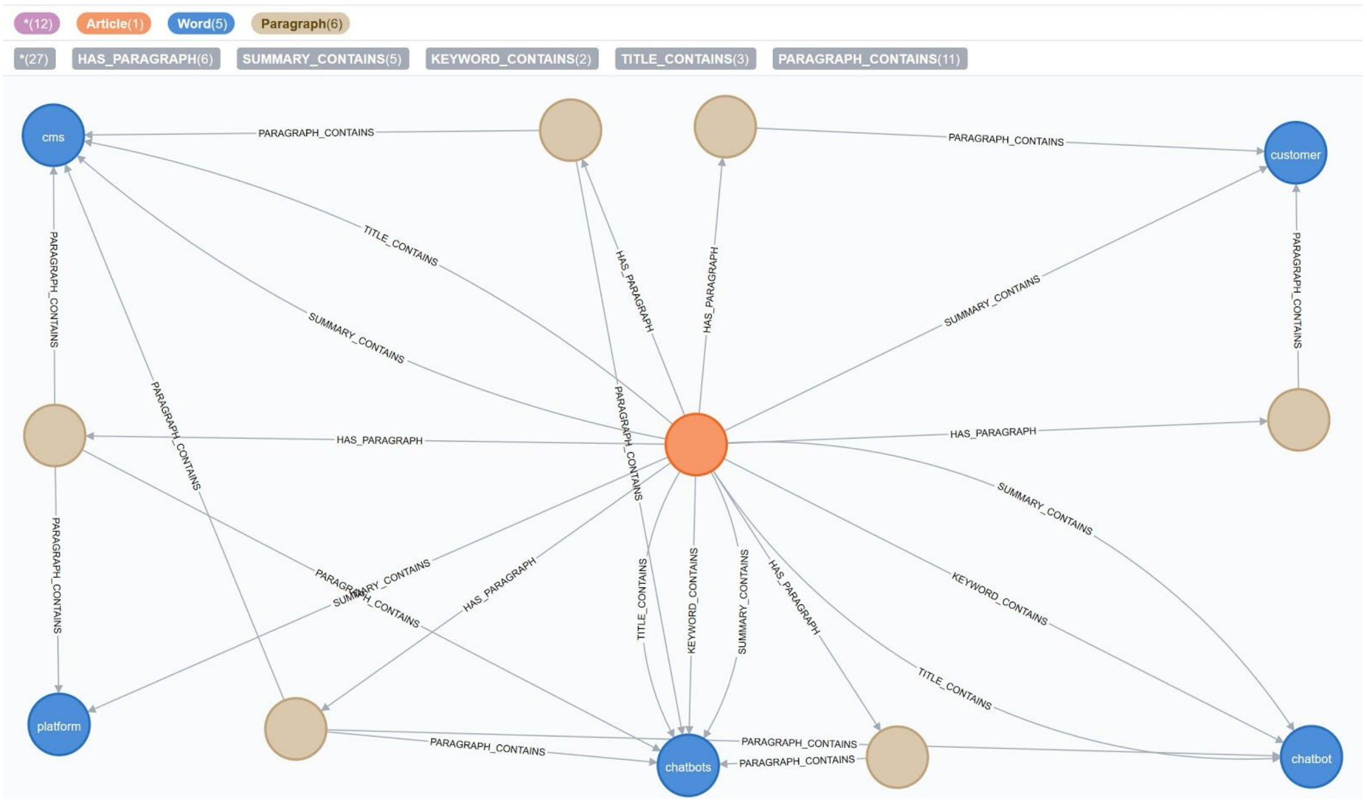


Figure 4.8: Showing an example of the data stored in Neo4j database as a graph format. Neo4j Browser is used to view this data.

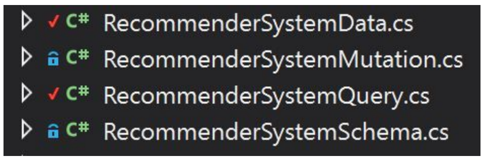
4.5 GraphQL API

We've come to the step where we have all the data, and now we want to make use of it. First, we'll begin by making it possible to make GraphQL queries to our web API project.

When using GraphQL for .NET we need to let the application know about a few things:

- **Schema**: this is the center of any GraphQL implementation and it describes the functionality available to the clients which connect to it.
- **Query**: here we define the data-fetching (querying) operations that can be executed by the client.
- **Mutation**: here we define the data-manipulation (mutating) operations that can be executed by the client.
- **Data**: Every GraphQL service defines a set of types that describe the possible data you can query on that service.

These four essential parts of the GraphQL API will be represented by its own class in C#, as seen in figure 4.9.

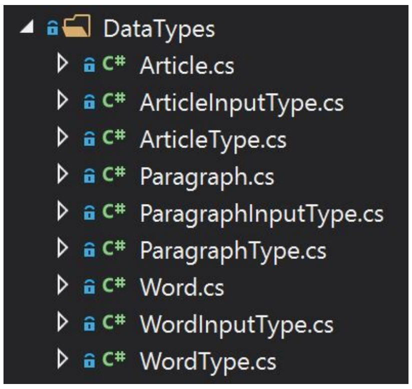


```
RecommenderSystemData.cs
RecommenderSystemMutation.cs
RecommenderSystemQuery.cs
RecommenderSystemSchema.cs
```

Figure 4.9: Classes that represent data, mutation, query and schema in GraphQL.

RecommenderSystemData class will be the middleman between C# objects and the data that mutations and queries are asking for. To handle this situation properly, it needs to have a way to turn C# objects into object graph types. We'll do this by having three types of data:

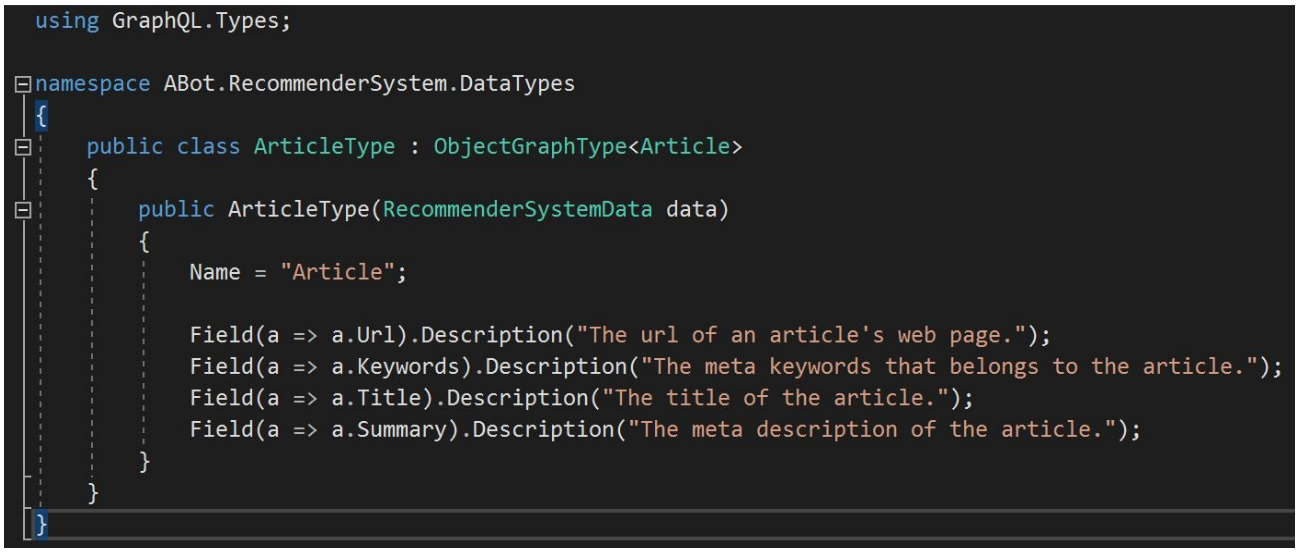
- C# classes
- ObjectGraphType<ObjectType> classes for handling queries
- InputObjectGraphType<ObjectType> classes for handling mutations.



```
DataTypes
    Article.cs
    ArticleInputType.cs
    ArticleType.cs
    Paragraph.cs
    ParagraphInputType.cs
    ParagraphType.cs
    Word.cs
    WordInputType.cs
    WordType.cs
```

Figure 4.10: Displaying C#, ObjectGraphType and InputObjectGraphType classes for all data types.

Figure 4.11 shows an example of how we might map a C# class to an object graph type class.



```
using GraphQL.Types;

namespace ABot.RecommenderSystem.DataTypes
{
    public class ArticleType : ObjectGraphType<Article>
    {
        public ArticleType(RecommenderSystemData data)
        {
            Name = "Article";

            Field(a => a.Url).Description("The url of an article's web page.");
            Field(a => a.Keywords).Description("The meta keywords that belongs to the article.");
            Field(a => a.Title).Description("The title of the article.");
            Field(a => a.Summary).Description("The meta description of the article.");
        }
    }
}
```

Figure 4.11: Connecting C# objects to graph types.

When the capability to make GraphQL queries is in place, we'll decide how we're going to score the relationships between the content nodes in order to find the best recommendation. What value will we give each relationship type to find the article that's the most likely choice to be a good recommendation?
And when an article has been found, we need to look for the highest scored paragraph in that article to provide as a reference for the recommendation.

This is the procedure we chose to follow:

1. Check if an article has all three relationships with a given word: TITLE_CONTAINS, KEYWORD_CONTAINS, SUMMARY_CONTAINS
    - If the article has all of these relationships, continue to step 2
    - If not, look whether it has at least one or two of these relationships.
        - If it does, continue to step 2.
        - If it does not, look for another article.
2. Check all the article's paragraph for if they have a PARAGRAPH_CONTAINS relationship with the given word
    - If a paragraph has that relationship, check for high count and frequency scores

An article that has all mentioned relationship types will have a higher recommendation score than an article with only one or two of the relationship types. We took the decision that if an article does not have the word in either its title, meta keywords or meta description - it's likely not a good recommendation unless it has a high score for both word count and frequency in a larger amount of its paragraphs. And as for the article's paragraphs, the one with the highest scores for both count and frequency will be returned.

Now, when we want to connect this setup to Dialogflow, we need the Dialogflow agent to first call our API and provide the word parameters, and then display the information received in its chat with the user.

To set this up, we enable webhook functionality in Dialogflow's dashboard where we also enter the URL of the recommender system's API together with basic authentication and headers - if needed. Then we create a fallback intent in Dialogflow where we activate webhook calls for this intent.
Finally, when a fallback intent situation happens, the recommender system will receive a POST request from the Dialogflow chatbot agent in the form of a JSON message. The request message contains data in Dialogflow webhook format, which

is an object containing information about context and parameters from the user's request. An example of Dialogflow webhook format object can be seen in figure 4.12.

```
{
   "responseId": "11bbffe6-6b19-4c43-8e67-f32a61dfc42a",
   “queryResult”: {
   "queryText": "Interesting. How do they fit together, chatbots and content engineering? I
         mean, how can I use a chatbot in my content engineering strategy?",
   "parameters": {
         "subjects": "chatbots",
         "subjects1": "content engineering"
    },
    "allRequiredParamsPresent": true,
    "fulfillmentText": "I'm sorry, I didn't catch what you meant.",
    "fulfillmentMessages": [
     {
         "text": {
            "text": [
               "I'm sorry, I didn't catch what you meant."
             ]
         }
      }
      ],
    "intent": {
         "name":
            "projects/small-talk-e234c/agent/intents/499329f1-72fb-46bb-9212-7ca049226663",
         "displayName": "Fallback intents",
         "isFallback": true
     },
    "intentDetectionConfidence": 1,
         "languageCode": "en"
     },
    "originalDetectIntentRequest": {},
    "session":
         "projects/small-talk-e234c/agent/sessions/1af87377-2afc-65fb-2ecd-aeb9ff46b494"
}
```

Figure 4.12: An example of the JSON message sent in Dialogflow webhook format.

5 Results

In the results chapter, the system evaluation will be presented. At first, we'll take a look at the technical evaluation aspects to determine if the system components have met the requirements. Then, we'll look at the system's ability to produce relevant recommendations in a use case scenario.

5.1 Evaluation based on technical requirements

By comparing the finished system components with the technical requirements specified in chapter 1.5, a compilation of the results has been constructed.
The system should:

Requirement	**Result**
Contain a web spider (crawler and parser) that fetches and parses content from one website at a time.	Fulfilled. The web spider component accomplishes these requirements.
Rank the content for specific subjects to tell how relevant each piece of content is to any given subject.	Fulfilled. The BoW model component processes the data to give each paragraph a count and frequency score for each word (topic).
Store the content and its rankings in a permanent storage.	Fulfilled. The graph database component is used to achieve this.
Grant API access to other system applications so that they can get and use the stored data.	Fulfilled. The graphQL component lets other applications use the system's API to find existing data.
Use .NET Core for backend development.	Fulfilled.
Present data in a graph format, displaying nodes and relationships between the content.	Partly fulfilled. When using Neo4j browser, the data can be viewed in a graph format but there is no exclusive component providing front-end visualization of the data.
Use Dialogflow's API to synchronize the system's graph data with a chatbot agent for testing.	Fulfilled. A fallback intent has been set up to use a webhook calling the recommender system's API.

5.2 Evaluation based on recommendations

The results (the outcome) produced by the recommender system is ultimately recommendations. The data, retrieved and parsed by the web spider which is then processed by the filtering algorithm, is simply stored in the database until the recommender system's API is used by another application.
There is no sole front-end component that uses this data regularly.

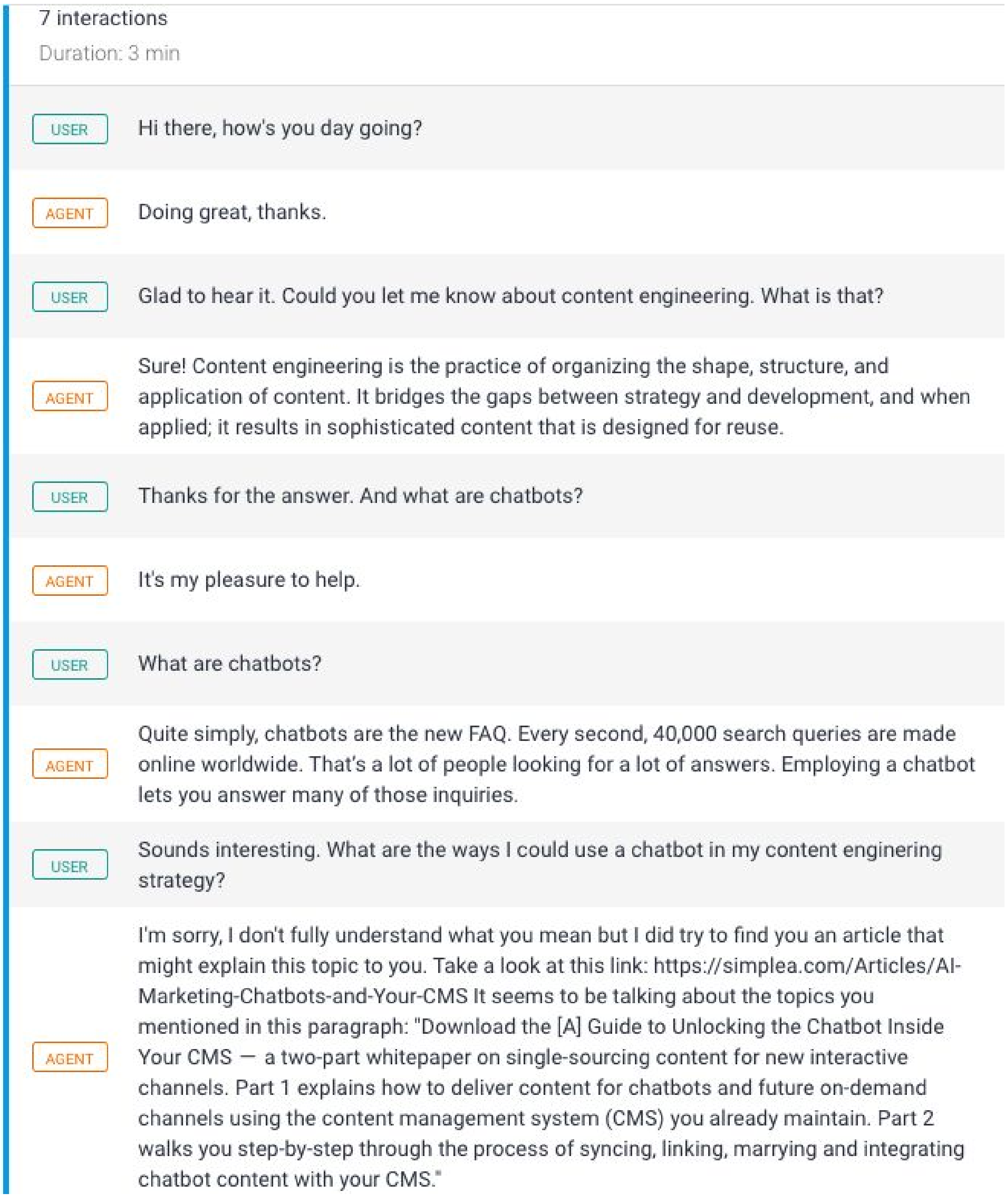
If we go back to the use case that was introduced in chapter 3.1.1, the user was requesting information about how to use chatbots in an content engineering strategy. The words picked up by the agent were "chatbots" and "content engineering", and they were sent as parameters to the API of the recommender system.
The recommender system found 3 matches for the word "chatbots":

1. Article (url: *https://simplea.com/Articles/AI-Marketing-Chatbots-and-Your-CMS*) has all three relationships to this word.
2. Article (url: *https://simplea.com/Articles/Chatbots-Content-Model*) has this word in its title and among the keywords.
3. Article (url: *https://simplea.com/Articles/2018-A-Content-Intelligence-Resource-Guide*) has this word in its meta description.

As for the word "content", the system found 30 article matches that have all three relationships to this word. That is most likely because this word is very common and can be used in a variety of ways. Therefore, we needed to check if any of these articles also have any relationship to the word "engineering". When using this approach, the count was down to 5 articles that have all three relationships to both the words "content" and "engineering".

The final step is for the system to filter these results depending on which articles that have relationships to all of these three words; "chatbots", "content" and "engineering". Unfortunately, there was none. As soon as the word "engineering" was included in the request - the system gave an empty response. It did, however, return a recommendation when the request only contained the words "chatbots" and "content".
From the article that the system sent as its final recommendation, it also found a paragraph with the highest score for both word count and word frequency of these two words to send with the response.

In figure 5.1 we can see roughly the same conversation that took place between a user and a chatbot agent earlier in figure 3.1. However, this time the chatbot answers with a fallback intent that gives some value and help to the user:


7 interactions
Duration: 3 min

USER Hi there, how's you day going?

AGENT Doing great, thanks.

USER Glad to hear it. Could you let me know about content engineering. What is that?

AGENT Sure! Content engineering is the practice of organizing the shape, structure, and application of content. It bridges the gaps between strategy and development, and when applied; it results in sophisticated content that is designed for reuse.

USER Thanks for the answer. And what are chatbots?

AGENT It's my pleasure to help.

USER What are chatbots?

AGENT Quite simply, chatbots are the new FAQ. Every second, 40,000 search queries are made online worldwide. That's a lot of people looking for a lot of answers. Employing a chatbot lets you answer many of those inquiries.

USER Sounds interesting. What are the ways I could use a chatbot in my content enginering strategy?

AGENT I'm sorry, I don't fully understand what you mean but I did try to find you an article that might explain this topic to you. Take a look at this link: https://simplea.com/Articles/AI-Marketing-Chatbots-and-Your-CMS It seems to be talking about the topics you mentioned in this paragraph: "Download the [A] Guide to Unlocking the Chatbot Inside Your CMS — a two-part whitepaper on single-sourcing content for new interactive channels. Part 1 explains how to deliver content for chatbots and future on-demand channels using the content management system (CMS) you already maintain. Part 2 walks you step-by-step through the process of syncing, linking, marrying and integrating chatbot content with your CMS."

Figure 5.1: Showing the fallback experience when the chatbot application is using the recommender system for support when responding the user.

6 Discussion

In this chapter, I will bring up three related topics of discussion. Firstly, I will discuss the system implementation and the results of this project. Secondly, ethical considerations that might be worthy of attention concerning the system and its functionality will be brought up for discussion. Thirdly, suggestions for future work and improvements will be considered.

6.1 System implementation

After reviewing the result, I have some thoughts I would like to bring up for discussion.

The advantage of splitting the recommender system into two projects is the modularity. It's easy to use the console application (web spider) whenever appropriate; it could be used only once or scheduled to crawl the website regularly at specific times. Just because another application is calling the web API does not mean that the web spider will be affected at all. It might even be possible to disable the data processing class from the console application and use the web spider for other purposes.

The guidelines and policies for web crawling that were implemented were considered to be necessary to follow in order to achieve the goal of this project. However, it's entirely possible to implement the remaining guidelines and policies as DotnetSpider library provides opportunities for fulfilling those as well.

A bag-of-words representation is simple to generate but far from perfect. All words are counted equally, and BoW only takes into consideration the frequency of single words in a document. This means that an article's important words will be words that occur a lot of times, which might not always be the case. When using BoW, we look for the quantity of the words, ignoring the order and context. To ignore the context means that we're ignoring the actual logical meaning of the words in the text.
Also, because we use only single words we're missing expressions. If we would look for bigrams instead, we would capture two words that have a meaning together, like "user experience" instead of "user" and "experience". A bag-of-bigrams representation could be much more powerful than our bag-of-words model.

There are also some tokens with the same word root but in different forms, e.g. "create", "created" and "creating". We saw this in figure 4.8 where "chatbot" and "chatbots" were different word nodes. In these cases, a normalization technique called stemming is required to degenerate different grammatical forms of a word to its root form.

As a last note, there needs to be a way to handle use cases were no match was found in the database. This was shown in the results when the words "chatbots", "content" and "engineering" were requested - the response returned empty. Luckily, we were able to get a recommendation for two of the words, but what if the user asks about a subject that this website hasn't written any articles about? Would it be appropriate then to use a generic fallback intent? And would we say that the recommender system did not support the chatbot fallback experience in that case as no extra value or help could be provided to the user?

6.2 Ethical considerations

There are several aspects to bring to attention when discussing ethical considerations of how this software system might be used. On the other hand, not all of these circumstances will be applicable when we think about how this system is being used in our use case.

The most prominent part of the system that could be used unethically is the web spider component. A web spider could be used to crawl web pages and extract copyrighted material, to find email addresses for spam lists, to put pressure on the server that runs the website - just to mention a few unethical behaviours.
In our case, the web spider component would not be handled like that. Its purpose is to be used on one website at a time, gathering information about recommendations to use for that website's chatbot application. It is, nonetheless, still an ethical consideration to discuss when developing web spiders.

Another aspect is that the recommender system could potentially be used with bad intentions; such as gathering information about business competitors' websites to either save/copy their copyrighted material or to analyze their content strategies.

This recommender system is also supposed to provide support to a chatbot application, which makes it relevant to discuss ethical considerations when it comes to chatbot development.
One of those crucial considerations would be to make sure that the chatbot is GDPR compliant if it's used inside the EU. The amount of personal data provided through a conversation with a chatbot might exceed the amount collected when the user fills out a form due to the nature of the conversation. Chatbot logs, for example, might be

a place where personal data such as IP addresses, e-mail addresses and names might be stored.

Another issue that's often heard of is the possibility of computers replacing humans by automating their work. This discussion also applies to chatbots as their purpose is to act as a form of customer service. But, automation does not have to be a bad thing. If chatbots can automate repetitive and straightforward tasks, it can free up time for the human workers to focus on more advanced cases. Users might also get quicker responses and easier access to customer service.

The last consideration I want to mention relates to accessibility. How could I make sure that other applications using this recommender system are accessible, allowing the recommendations to be used by everyone?

6.3 Future work and improvement suggestions

Time has been limited during this project, and therefore, there are both improvements to be made as well as future work to be done in the form of testing and development of features and functionality.
Below are some thoughts and ideas.

The evaluation tests made on the recommendations were limited to a small amount. Tests on a larger volume would be needed to see how the results scale.

One suggested improvement is to include some library for machine learning (ML) models, either replacing or supporting the scoring algorithm component of the system. That would be interesting to implement because I think we would see a raised quality of the recommendations when it comes to how relevant they are for the context of the dialogue.

The purpose of this recommender system was to support a chatbot application in case of fallback intent, but it would be interesting to see if the system works with other applications and in different scenarios. One suggestion might be crawling through resumes for a job application and see if the recommender system could provide some insights into which candidates that would be a possible fit, based on the subjects they have mentioned in their resumes.

A front-end component could be built for the system to provide its human users with different choices of controlling the application, like a selection of options for websites to crawl, and also a way to display the results that are stored in the database in a graph and/or table format.

The system is currently only recommending articles, but it would be of interest to see if other types of content like video, images and infographics could be used as recommendations as well.

If there are several recommendations with the same score in relations to a specific topic, another feature for the system could be to measure how often the same recommendation is shown to the user. Does it happen that users keep seeing the same recommendation repeated times? Could some randomization in choosing the right content to recommend be useful at some point?

What if previous user behaviour could influence the recommendations. Looking at cookies saved from earlier visits or history from previous chats as well as other information there is about a user like what web pages he/she has visited before, and so on. Could data like that be sent with Dialogflow webhook as parameters?

None of the metrics I've mentioned above matters more than how real users react to the recommendations. Evaluating the results with real users would be of great value. This could be done by the chatbot agent asking the user for explicit feedback in the chat of how she would rank the recommendations that were suggested. This is called to measure the perceived quality of the recommendation, and if an ML model would be included in the system - this could be used to train that model into learning and improving its recommendations.
I would also strongly suggest doing user testing to find out how people will interact with the system.

7 Conclusion

The purpose of this independent degree project was to investigate to what extent a content-based recommender system can be used to support the chatbot fallback experience.

Can a recommender system be used to improve the chatbot fallback experience?

Yes. It is possible to build a recommender system that will rank content and produce recommendations for a chatbot application to use in case of fallback intent. This would improve the chatbot fallback experience to the extent that instead of the agent answering with a random generic response, it could provide some vital value to the user that might help on his/her way to find the answer.

However, I want to draw attention to the fact that a recommender system would not be a good fit in all circumstances of fallback intent. It would improve the chatbot fallback experience in use cases like the one mentioned in this report where the user is looking for information or general answers of some sort.
But, in scenarios where the use case is more task-oriented, e.g. a user might want to place an order or get in contact with a human for support, this system might not be a favourable choice.

Will the recommender system's scoring algorithm suggest content that is relevant to the user's input?

The current scoring algorithm used by the system will not take the context of the dialogue into consideration when making recommendations. There is, however, a possibility to advance the algorithm to a stage where the context would be taken into account, as mentioned in chapter 6.3 about future work and suggested improvements. The recommendations would, in that case, be entirely relevant to the user's input within the context of the dialogue.

In conclusion, the evaluation tests have shown some promising strengths of the implemented system but also limitations. Our current system can be used as a support to provide a chatbot agent with article recommendations successfully. The recommendations, however, are based on the basic foundation of word count, i.e. how many times a word appears in the article text. The system could achieve significant improvements in its recommendations if a ML mechanism was introduced as it would use a more advanced scoring algorithm.

References

[1] "Google, Microsoft And Startups Are Going To War On Chatbot Technology", *Forbes.com*, 2019. [Online]. Available: https://www.forbes.com/sites/parmyolson/2018/07/27/google-microsoft-and-startups-are-going-to-war-on-chatbot-technology/. [Accessed: 30- Mar- 2019].

[2] "Dialogflow", *Dialogflow*, 2019. [Online]. Available: https://dialogflow.com/docs. [Accessed: 30- Mar- 2019].

[3] A. Shevat, *Designing bots*, 1st ed. O'Reilly Media, 2017.

[4] C. Phillips, "Perfecting the Chatbot Fallback Experience", *UX Planet*, 2019. [Online]. Available: https://uxplanet.org/perfecting-the-chatbot-fallback-experience-f76d119c45d4. [Accessed: 01- Apr- 2019].

[5] F. Isinkaye, Y. Folajimi and B. Ojokoh, "Recommendation systems: Principles, methods and evaluation", *Egyptian Informatics Journal*, vol. 16, no. 3, 2015.

[6] D. Wang, Y. Liang, D. Xu, X. Feng and R. Guan, "A content-based recommender system for computer science publications", *Knowledge-Based Systems*, vol. 157, 2018.

[7] "Transforming Digital Content and CEM", *[A]*, 2019. [Online]. Available: https://simplea.com/Home. [Accessed: 05- Apr- 2019].

[8] C. Barrett, "The Frustration of Irrelevant Content", *CodifyMedia*, 2017. [Online]. Available: https://codifymedia.com/frustration-irrelevant-content/. [Accessed: 10- Apr- 2019].

[9] "Understand how structured data works", *Google Developers*, 2019. [Online]. Available: https://developers.google.com/search/docs/guides/intro-structured-data. [Accessed: 30- Apr- 2019].

[10] C. Castillo, "Effective Web Crawling.", Dept. of Computer Science - University of Chile, 2004.

[11] M. Thelwall and D. Stuart, "Web crawling ethics revisited: Cost, privacy, and denial of service", *Journal of the American Society for Information Science and Technology*, vol. 57, no. 13, 2006.

[12] D. Sullivan, *Advanced NoSQL for Data Science*. https://www.linkedin.com/learning/advanced-nosql-for-data-science: LinkedIn Learning, 2017.

[13] "An efficient recommender system based on graph database", *Kernix.com*, 2019. [Online]. Available: https://www.kernix.com/blog/an-efficient-recommender-system-based-on-graph-database_p9. [Accessed: 22- Apr- 2019].

[14] C. Liu, R. Lowe, I. Serban, M. Noseworthy, L. Charlin and J. Pineau, "How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation.", School of Computer Science, McGill University., 2017.

[15] "dotnetcore/DotnetSpider", *GitHub*, 2019. [Online]. Available: https://github.com/dotnetcore/DotnetSpider. [Accessed: 30- Apr- 2019].

[16] "NuGet Gallery | Home", *Nuget.org*, 2019. [Online]. Available: https://www.nuget.org/. [Accessed: 30- May- 2019].

[17] "XPath Tips from the Web Scraping Trenches", *Hacker Noon*, 2019. [Online]. Available: https://hackernoon.com/xpath-tips-from-the-web-scraping-trenches-fda06b0bf0a8. [Accessed: 04- May- 2019].

[18] J. Brownlee, "A Gentle Introduction to the Bag-of-Words Model", *Machine Learning Mastery*, 2017. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-bag-words-model/. [Accessed: 07- May- 2019].

[19] "Neo4j Graph Platform – The Leader in Graph Databases", *Neo4j Graph Database Platform*, 2019. [Online]. Available: https://neo4j.com/. [Accessed: 10- May- 2019].

[20] "neo4j/neo4j-dotnet-driver", *GitHub*, 2019. [Online]. Available: https://github.com/neo4j/neo4j-dotnet-driver. [Accessed: 12- May- 2019].

[21] "GraphQL: A query language for APIs.", *Graphql.org*, 2019. [Online]. Available: http://graphql.org/. [Accessed: 16- May- 2019].

[22] "graphql-dotnet/graphql-dotnet", *GitHub*, 2019. [Online]. Available: https://github.com/graphql-dotnet/graphql-dotnet. [Accessed: 13- May- 2019].

[23] "Visual Studio IDE, Code Editor, Azure DevOps, & App Center - Visual Studio", *Visual Studio*, 2019. [Online]. Available: https://visualstudio.microsoft.com/. [Accessed: 14- May- 2019].