**Bulgarian Diploma Thesis**

**ChatAUBG**

**Stefan Leon, ID#200158170**

**Student:**  **, Date:**

*signature*

**Supervisor:**  **, Date:**

*signature*

**Department of Computer Science, AUBG**

**Blagoevgrad, 2024**

**Table of Contents**

1. **Introduction**3
2. **Specification of Software Requirements**4
3. **Design of Software Solution**9
4. **Implementation**22
5. **Testing**42
6. **Results and Conclusion**50
7. **References**

**1. Introduction**

The field of Artificial Intelligence has received a great deal of attention and scrutiny recently; specifically, with the explosive rise to popularity of OpenAI’s ChatGPT (<https://chat.openai.com/>), chatbots – a term generally referring to computerized text-processing systems, meant to replicate human-to-human conversation – have become ubiquitous: on Facebook Messenger alone, the number of bots has quadrupled from 2017 to 2020, from 100,000 to an impressive 400,000 [1]. Similarly, many businesses have been investing in developing and incorporating chatbots into their operations, as a highly controllable, cost-effective approach to customer service [2].

Personally, I was impressed by the immense versatility of many multi-purpose chatbots; the way people could not only talk to them, but seemingly connect to something that is nothing more than a stochastic language model was incredibly interesting to me. As such, I began to look into Natural Language Processing tools and techniques that people use to make interactive systems, and how designers go about shaping the user experience. This loose process of research allowed me to accrue more and more knowledge about the field, eventually culminating in this Senior Project.

According to a systematic review of literature pertaining to chatbots [1], there are two big categories of systems that currently dominate the field: there are those that use text-based channel to converse with a user, usually making the conversational aspect the ultimate aim; and there are those that act as virtual assistants, helping the user with certain tasks, depending on the natural language input fed into them. Most chatbot systems land somewhere in between these two categories, making use of conversational capabilities, while also having the ability to complete some sort of task for the user – as does my project.

The chatbot I developed – to be referred to as ChatAUBG from here on out – serves as a virtual assistant for users trying to browse the American University in Bulgaria’s (AUBG) website (<https://www.aubg.edu/>). Its aim is to help guide users through the various pages on the website, and to help provide them with relevant information pertaining to the questions they might have. In terms of classification, my ChatAUBG would be categorized as a Question Answering System.

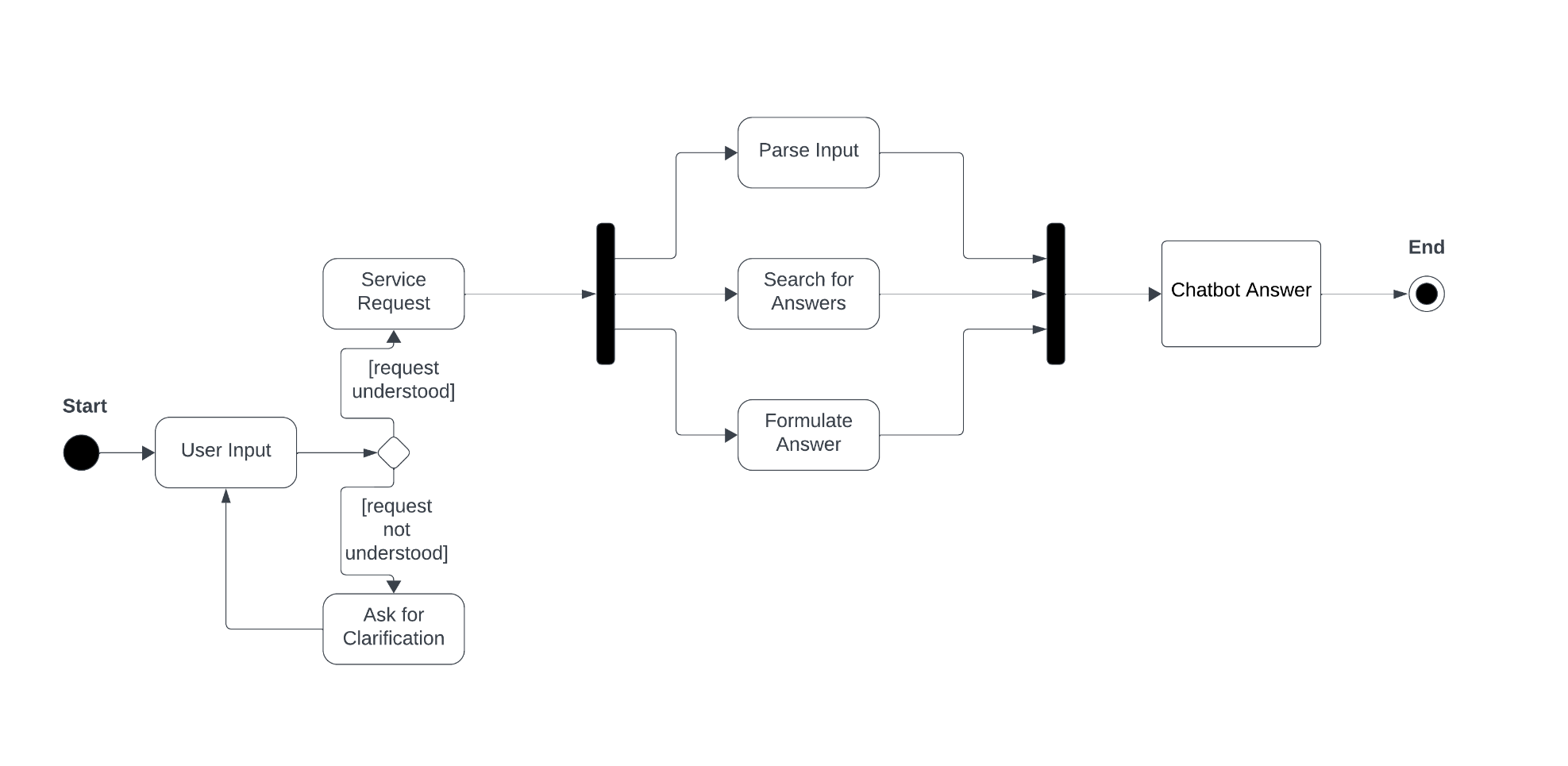
The language I chose for developing this project is C++, due to its malleability and powerful capabilities. I will further elaborate on this choice in section 4. Implementation.

**2.** **Specification of Software Requirements**

Broadly, Question Answering Systems (QAS) are defined as applications whose purpose is to take in and process a user’s request, and to subsequently retrieve that information from a knowledge base, and reply to the user’s request. What makes them different from normal search engines is their ability to respond to queries with specific, targeted responses, without extraneous information – like a web search engine would, for example, when replying to a user query with a list of possibly relevant webpages [3]. There are three main functions that such an application must fulfill, each of them important for assuring user satisfaction.

A diagram of a chatbot

Description automatically generated*Figure 1. UML component diagram, representing the discrete functions that the Chatbot’s software must fulfill, as well as the flow of information between the components fulfilling each of these functions, in a typical user interaction. Based on terms mentioned in relevant research [3].*

**

*Figure 2. UML state diagram, representing the application flow of ChatAUBG, from the moment the user first inputs some request into the chatbot’s interface, and until the program formulates a response. A notable branching point is at the moment of attempting to parse the input: if the chatbot cannot understand the user’s request, or map it onto any possible response, it will ask for clarification instead of continuing its execution as usual.*

Firstly, there is the matter of Question Analysis [3]. The chatbot must be able to parse and understand the user’s input, in order to extract relevant information about the query it must respond to. As computerized systems cannot understand the meaning of language in the same way a human can, the input must be processed into something the chatbot can understand, and Natural Language Processing techniques must be employed to achieve this.

Secondly, the application of must have some component that can perform a Search of Documents [3]. The chatbot has to figure out where the information pertaining to the user’s question can be found, and fetch enough of that information in some manner. This function is similar to the one that an average search engine fulfills, and this component could be allowed to function in a similar way to search engines.

Thirdly, the chatbot must perform the Extraction of an Answer [3]. The application must know what to do with the information it accesses and subsequently process it into a form that would be understandable for an average user. This step will once again likely involve the use of advanced Natural Language Processing techniques, that allow for limited textual generation, and which will contribute to the conversational ability of the application.

Considering my project is aimed at fundamentally improving the user experience of the AUBG website, I also considered whether I could get real user feedback, to base my software design on. I reached out to the Marketing Office of AUBG, and exchanged several emails, as well as having an in-person meeting with a representative. At the end of it, I was unfortunately unable to acquire any real data to base my decisions on – despite the representative’s excitement regarding my project – and so I will have to design my project’s features based on current trends and general concerns of QASs.

In fostering a more realistic environment, I would have ideally preferred to run over basic problematic areas of the website, and figure out where my project could intervene and what it could improve on. However, due to the critical lack of data, I will target the following areas for improvement:

1. Lowering the time spent searching for information on the website;
2. Providing a natural language solution for searching for information on the website;
3. Improving user satisfaction via natural responses to a user’s query.

I will now elaborate on the above-mentioned areas for improvement. In terms of lowering the average time spent searching the website for particular factoids, my solution provides the user with an alternative to regular searching algorithms – which require the appropriate and skillful use of keywords, which many users are not privy to. Instead of having to navigate the ins and outs of keywords – where the keywords inferred by the users need to match those used by the AUBG website – the users can address the chatbot in natural language, describing what it is they are looking for, and subsequently receive an answer. This should limit the time spent browsing across webpages in search of information.

Providing the user with natural language-based search algorithms should in turn also increase the quality of the user’s experience in interacting with the website, by allowing them to converse in natural language. In addition to the time saved by reaching the desired information faster, they can also do so in a more human-like manner.

Finally, providing the user with natural language responses should further enhance user satisfaction. Instead of having to look over several webpages that might ultimately be irrelevant to them, the user will simply be provided with relevant information, delivered in a natural, human-like medium.

**3. Design of Software Solution**

When it comes to developing chatbot software, there are two broad approaches one may take: the first – referred to as rule-based – relies on a predefined propositional structure, and often on a hard-coded set of reply types, that the chatbot chooses between, while mapping the user’s input text on one of the several scenarios it has access to. The second approach – referred to henceforth as corpus-based – relies on a dataset of natural language, of ideally large proportions, from which it will extrapolate how to formulate replies. While the two approaches are not necessarily exclusive in nature – indeed, the best software incorporates some elements of both approaches – it became apparent early on in my project that I would have to decide to focus on one or the other, when facing the time limitations.

I initially favored a rule-based approach. I am more familiar with the development of such systems: I previously looked into the first chatbot ever developed, ELIZA [13], and attempted to reproduce something similar to it. However, due to the complexity requirements of this project, it quickly became clear that a pure rule-based approach would not be appropriate, as the difficulty there consists in constructing a good enough set of predetermined reply conditions to simulate real conversation. It would also be very difficult to scale, as each new piece of information one would offer the chatbot would likely require at least one new reply case, making the work of maintaining and updating the software extremely labor-intensive. Thus, I settled on a mostly corpus-based approach; it is important to note, however, that certain hard-coded rules will become important, in order to tackle challenges when it comes to purely statistic textual generation.

As mentioned in the Introduction, the program is sectioned into 3 main components: Question Analysis, Search of Documents and Answer Extraction. I will now elaborate on the main subcomponents of each of these sections, as motivated by a review of common techniques used in constructing chatbot software [4].

The “Question Analysis” component is made up of various Natural Language Processing algorithms. Firstly, I created a basic tokenizer (defined as *tokenize*), that receives a string of words as input, and breaks down the string into individual words (as separated by a blank space), and subsequently outputs a vector of words, corresponding to the original input string. I also made use of a custom function for converting a string to its lowercase equivalent (defined as *to\_lowercase*), that converts each of the string’s individual characters to its lowercase counterpart. Additionally, I created another function for smoothing (defined as *smoothe*) that receives an input string for processing, and returns that same string, without its separators, punctuation marks and without non-English characters – which would significantly complicate the range of words my project would need to consider, given that the AUBG website is partly in English and partly in Bulgarian. Specifically, ChatAUBG will only handle input in English, and only be able to reply in English.

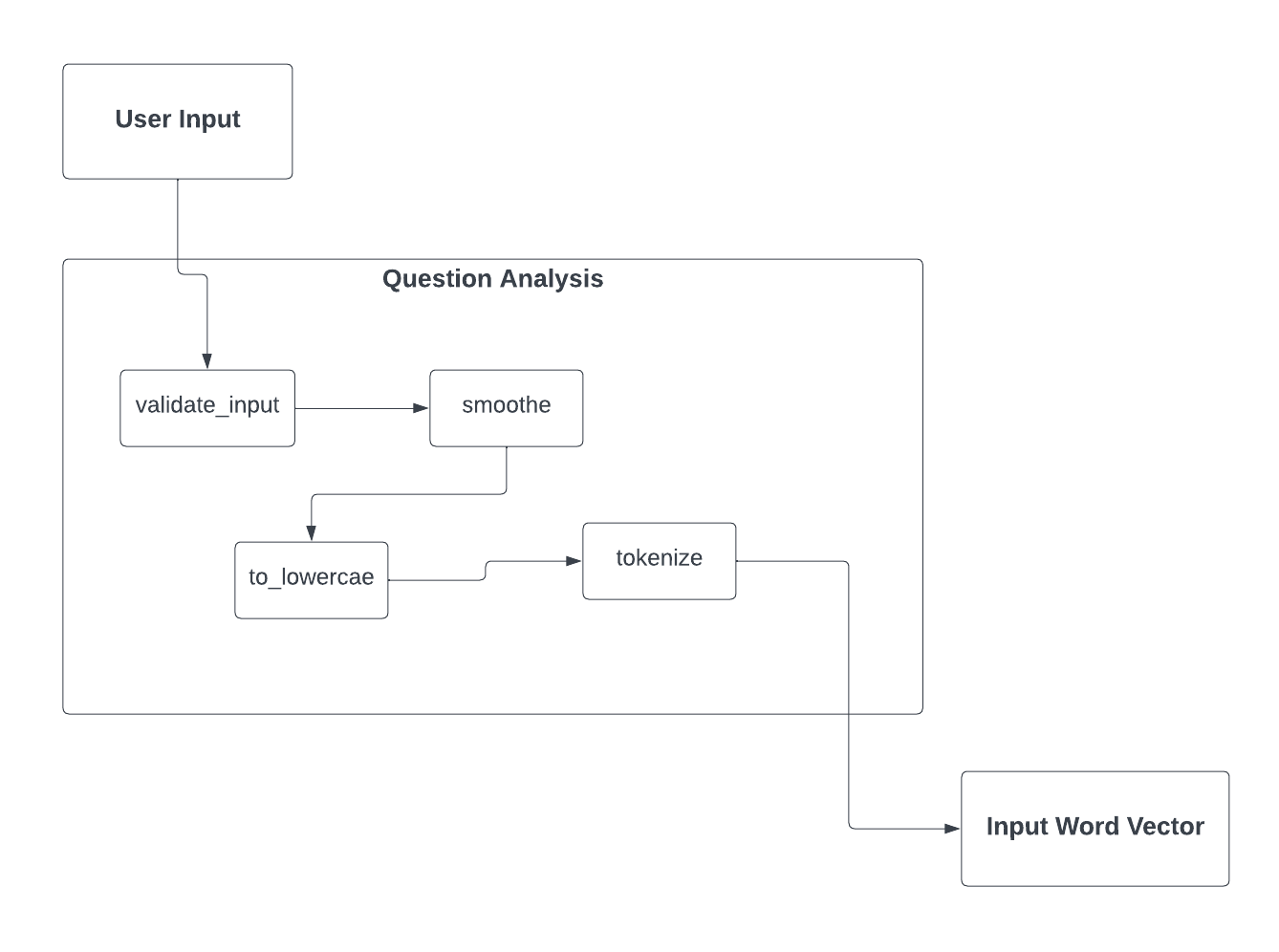
Additionally, I created a function called *validate\_input*, which, as the name suggests is tailored to validating the user input. In particular, it is especially relevant for deciding whether to move forward with the execution of the program, or to ask the user a clarifying question, in case of improper and incorrect input – for example, input that is not in English.

These functions are essential for parsing the user’s text into a proper form, so that the chatbot can understand it and match it against the information in its knowledge base. Furthermore, this helps compensate for certain common human typos, such as writing “dont” instead of “don’t,” or forgetting to capitalize a person’s name, by treating both the *correct* and *incorrect* versions of the word the same, after smoothing.

A screen shot of a question analysis

Description automatically generated

*Figure 3. UML component diagram depicting a further breakdown of the Question Analysis component of ChatAUBG. The logical relationships between the subcomponents is also depicted: the input string is first converted to lowercase; then, the new string is fed through the smoothing algorithm; finally, it is fed through the tokenizer function. The result will then be utilized for fulfilling the Search of Documents function.*

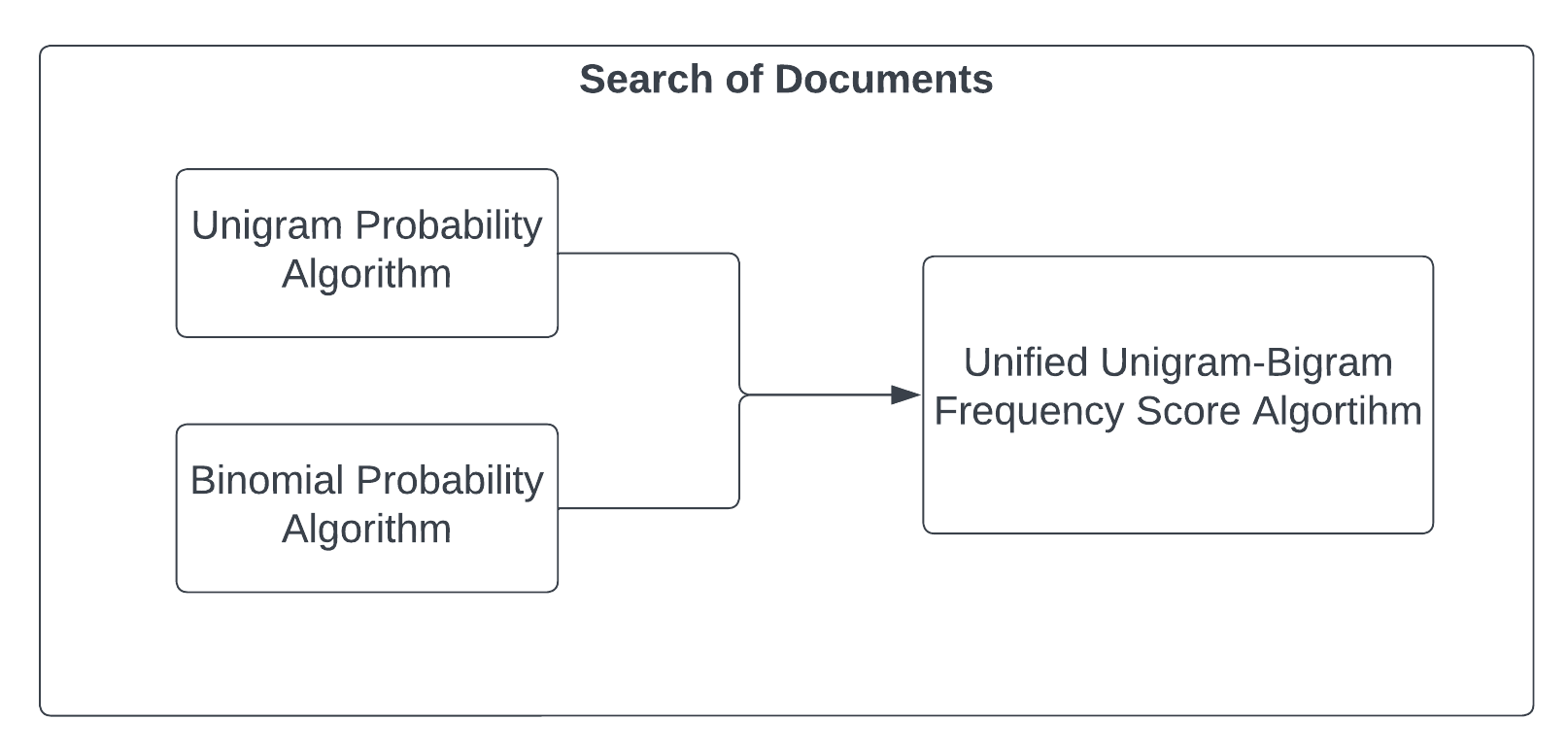


*Figure 4. UML component diagram, representing the un-abstracted breakdown of the Question Analysis part of the project, as it is implemented in the source code. The functions used for completing the tasks depicted in Figure 3 are shown, as they are named in the source code.*

To achieve an effective “Search of Documents” component, I initially considered implementing a normal search function, that would simply scan the available webpages for keywords and return a set of possible locations that would contain information pertaining to the user’s request. However, after working on implementing one for some time, I decided I could improve on it, by implementing a combined unigram and bigram model of text classification, to decide on the appropriate location of the information the user is looking for.

A unigram, at its core, is a text classification algorithm that – using the frequency distribution of the words fed to it as input – can determine the probability that a word is found in a certain category [5]; furthermore, using some assigned weights and some separate algorithm for computing, it can determine the probability of a set of words – a sentence, or, in my case, a string of words representing the user’s request – being in any one of n categories.

As per [5], there are several ways to increase the efficiency and performance of a unigram-based algorithm. Firstly, it is statistically biased towards negative categorization results – i.e., by traditionally assigning a probability of 0 to a word that does not appear in a category’s initial training dataset, the dataset will excessively underestimate the likelihood that a sentence containing that particular word can fall into that said category. Thus, to increase the performance of this algorithm, it is advised that we account for this, by coding the absence of a word in a dataset as a very small value, but still above 0 [5]. In my project, I did so by assuring that, upon computation, words that are not encountered in the dataset would still have a base frequency of 0.0001, instead of 0; this, being still significantly lower that that of any word that appears in the dataset – which would be at the very least 1, and most likely a significantly larger integer value – helps the unigram distribution work better, and makes my algorithm more effective.



*Figure 5. UML component diagram, depicting the breakdown of the Search of Documents component of ChatAUBG. Two separate probability algorithms are executed: a unigram-based one, that computes the probability of any one word in the now-processed input string being on any one webpage on the AUBG website; and a bigram-based one, that computes a similar probability, while computing a different probability for each word, depending on the word coming before it. Finally, these two probability scores are unified into one that takes into account both results.*

I also implemented a bigram algorithm, to further improve the search function further. A bigram works similarly to a unigram, in the sense that it can compute the probability of a sentence fitting in certain categories based on the words it is comprised of. However, instead of considering each word by itself, like the unigram algorithm, a bigram takes each two-by-two pair of words in the sentence and considers their probability of fitting in any category. Being a variant Markov chain algorithm, the bigram probability of any word will vary based on the word that comes before it; thus, the algorithm assigns greater importance to the particular context in which any one word is encountered in, something the unigram model is unfortunately blind towards.

The idea behind implementing both a unigram model and a bigram model for classification stems from research [6] showing that a mixed approach that considers both scores can improve performance on natural language processing tasks like feature extraction. This should safely generalize for the classification functions of my algorithms as well, and thus improve the Search of Documents performance of ChatAUBG as well.

Thus, using the aforementioned algorithms, ChatAUBG computes the probability of a certain – now processed – user input string being in any one of the webpages of the AUBG website. It first computes the unigram probability of the input string, followed by the bigram probability of it, and finally unifies them into one, using an importance algorithm, offering more weight to the pair frequencies discovered while computing the bigram probabilities. Classifying the request sentence into one webpage should also give us the webpage on which the answer to the user’s query can be found; this will be useful for the final phase of the chatbot’s processing, the Answer Extraction.

Essentially, there are three components in the Answer Extraction part of a ChatAUBG reply: first, there is a trained Markov Chains artificial intelligence model that computes and stores the transition probabilities between words, as they appear in the dataset. Secondly, there is a biasing algorithm, that, using the solution we find in the Search of Documents step of the program, would dynamically modify the Markov Chain model, to better reflect the particular contents of that webpage, and offer better targeted information to the user. Finally, the chatbot would then create a probabilistic textual reply for the user’s query.

There are several concerns to consider here: firstly, the length of the reply text needs to be somewhat consistent across replies. Because proper language knowledge – such as part of speech or part of sentence tagging, or knowledge of propositional structures like syntax and semantics [10] – is outside the scope of this project, ChatAUBG needs a different mechanism for determining when and how to end a reply. I ultimately decided to rely on absolute character length, in combination with punctuation marks – such as a period, question mark or exclamation mark – to decide when to end a sentence and when to end a particular reply. While basic, this should serve as a quick and effective solution to providing relatively consistent answers, in terms of length.

In order to train all of the models needed for my program, I first needed to create an appropriate dataset. Given the purpose of my project – creating a chatbot for the AUBG website – I thought it would be only fitting to obtain the dataset off of the website as well. As such, I decided to rely on text data currently available on the aubg.edu website, as it is described in the sitemap of the website. I will elaborate on the particularities of how I constructed this dataset further in chapter 4. Implementation. For now, it will suffice to say that I compiled all of this text data into a CSV (Comma Separated Values) file, called “aubg\_map.csv”

A screenshot of a computer

Description automatically generated

*Figure 6. UML component diagram, depicting the breakdown of the Extraction of Answer component of ChatAUBG, in three steps. Firstly, the Markov Chain model is computed, using the entire document to calculate the probability of any particular word coming after any other particular word. Then, this algorithm is biased using the particular word choices that appear in the webpage containing the data the user is looking for. Finally, a textual reply is generated and outputted for the user to see and reply to.*

The unigram and bigram models both rely on a Naïve Bayes algorithm, that treats the collection of text as a loosely connected “bag-of-words” [11]. The particular mathematical method for getting the probability of a word – or pair of words, respectively – consists of counting up the number of times the particular word appears in the class, and dividing that number by the amount of words – or pairs – in said class. The probabilistic classification of a sentence is then compiled by multiplying the probabilities of every word – or pairs of words – in that sentence appearing in any of the considered classes.

An important consideration, then, must be given for the case in which a word never appears in a given class – that is, its probability of appearing in that class is 0. In that case, the Naïve Bayes probability of any sentence containing that word to belong to that class would always be 0, irrespective of any other words in it. This is, predictably, a large issue that ChatAUBG must tackle somehow. In order to resolve this, a commonly used method is Laplacian smoothening, which essentially consists of assigning any particular word a minimal probability to appear in any of the considered classes, to avoid invalidating an entire class via a single word [11].

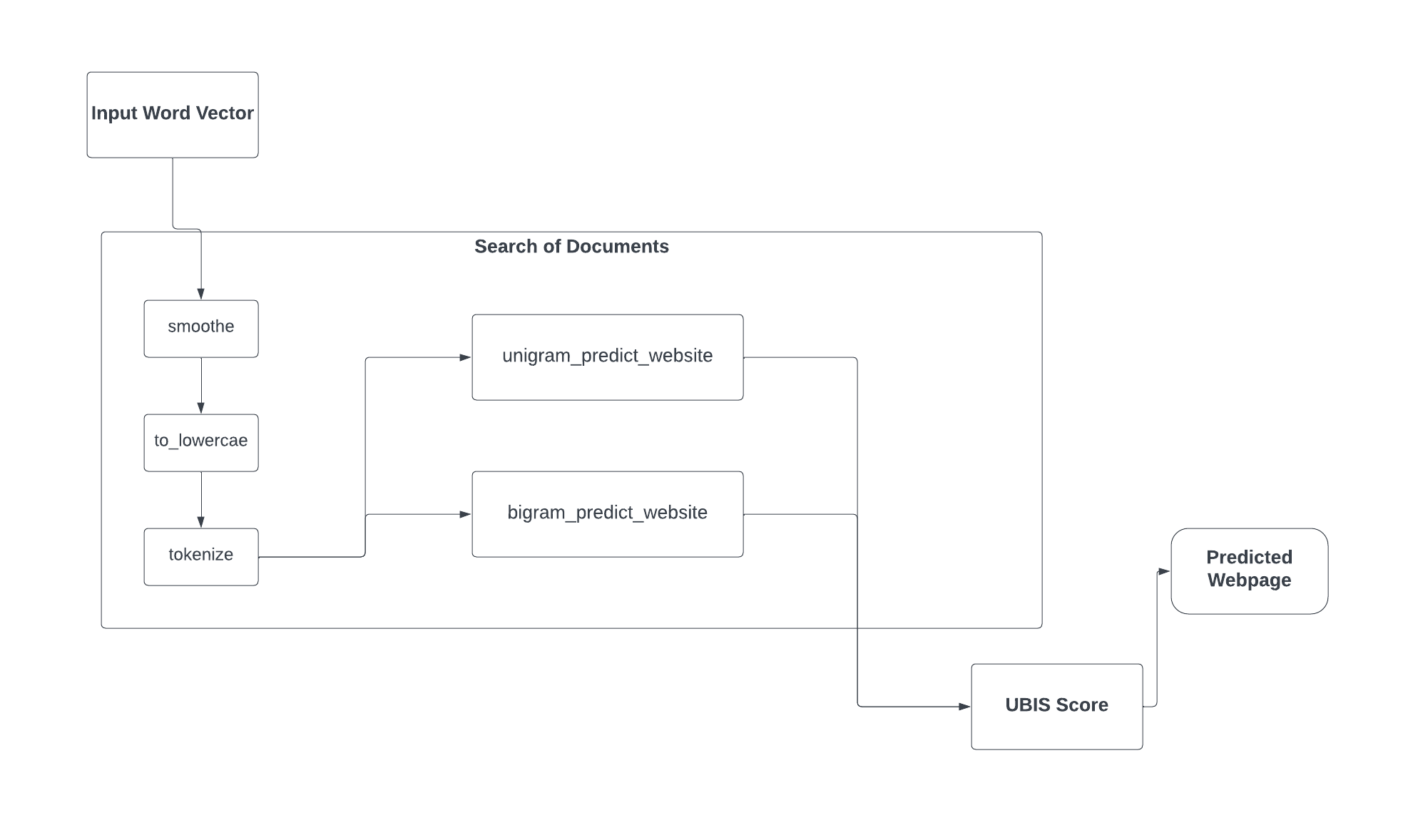
The probability used must be sufficiently small not to affect the probabilities computed by the models too much. Most simply, you might just calculate the probabilities by considering every word appears at least once in any given class. However, in my particular case, I noticed that the probabilities I was working with were sufficiently small that a difference of 1 affects the model excessively. Thus, I settled on a smaller value – particularly, 0.0001 – as a default with which to initialize all the probabilities associated with a class that does not contain a particular word. Furthermore, I used this value as a default value for new words – or pairs of words – I would have to dynamically add to the models and consider, as they are used in user input.

Finally, both models would be used in tandem to predict what webpage a user’s query would fit into, and compute two important variables: a *score* variable, that would average out the resulting unigram and bigram probabilities, and the address of a website– stored in memory as *max\_key*, as it refers to the class key of the maximal probability value – which would then be passed on to the next part of the program, the Answer Extraction.

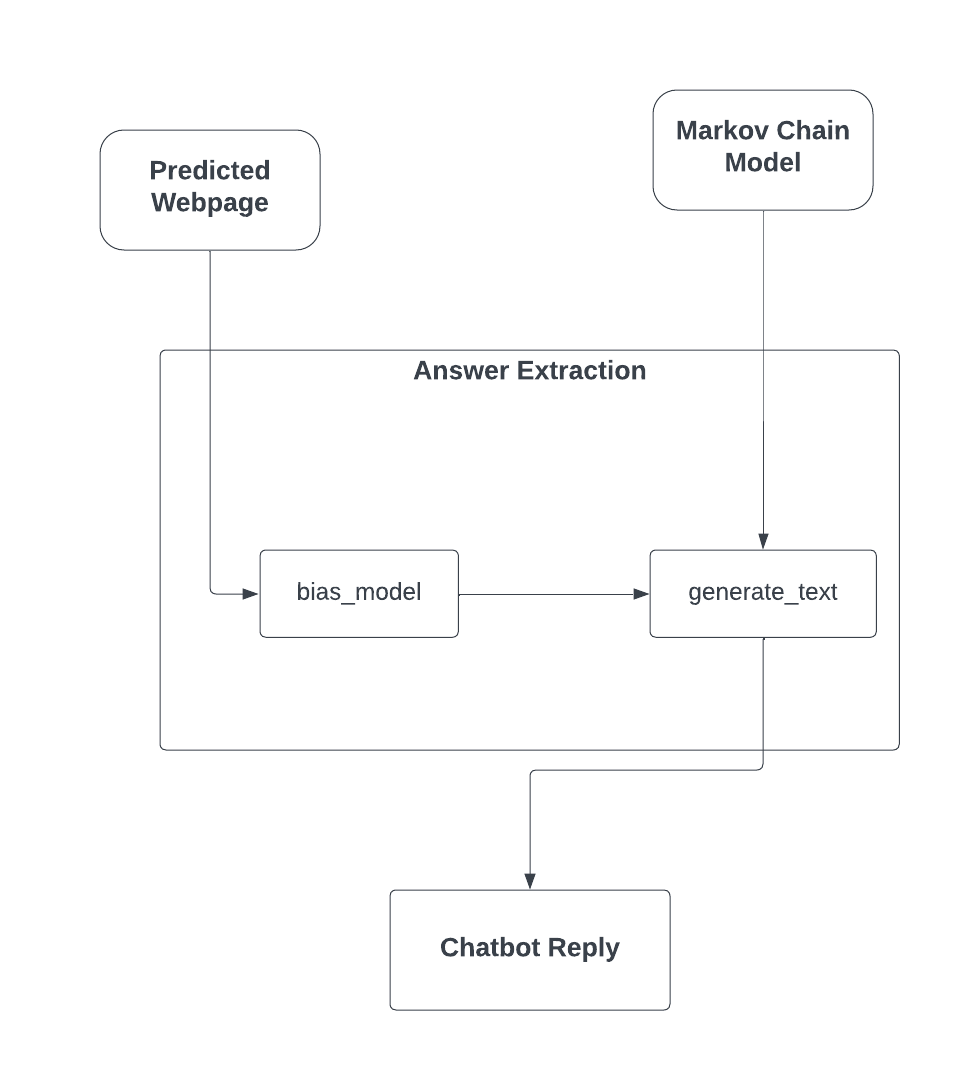
A further breakdown of the second part of the program is illustrated in Figure 7.

Finally in order to implement the Answer Extraction component of my program, I set out to implement a Markov Chains-based model of text generation. A Markov Chain artificial intelligence model essentially relies on discrete program states, and on a transition matrix detailing the probability of transitioning to another state, from any of those given states. In a Natural Language Processing context, they are often used for stochastic textual generation [12]: the transition matrix is particularly suited for computing the probability of any one word occurring after any other word.

Such a model would be capable of learning from an – ideally large enough – corpus how which words should come after which, and reproduce text in this manner, without actually having to possess a proper formal understanding of language, structure or syntax [10, 12]. This fits with my overall theme for this project, and can benefit significantly from the dataset I invested so much time into. Thus, it would make the most sense to implement a generative algorithm according to these criteria. As mentioned before, in addition to this well-researched, industry-standard approach, I will also be implementing a biasing algorithm, that would serve to further improve the quality of replies the chatbot will be able to output.



*Figure 7. In-depth UML component diagram, representing the real components of the Search of Documents part of the chatbot, as defined in the source code. The functions here rely on both functions from the Question Analysis part, and on the resulting input word vector, previously depicted in Figure 4.*



*Figure 8. UML component diagram, further detailing the functions and data structures involved in the Answer Extraction part of the chatbot. The names and relationships illustrate the way these functions are coded, and better explain how the tasks involved are resolved.*

It is likely also worth noting that, given the nature of my program, I did not have to grapple with too many security considerations. In fact, my chatbot hardly opens any room for security vulnerabilities, at the moment: should it be implemented on the AUBG website, it still only has access to publicly accessible text data, that is merely a few well-placed clicks away. What it offers the user is a comfortable and time-effective alternative to searching for information by themselves. Should it be developed to encompass additional functionalities within the website – basic payment processing, for example, or access to information unlisted on the public website – security concerns might slowly appear, and further encryption of this data could be implemented.

Similarly, the nature of conversation between a user and the chatbot, by its current nature, hardly incentivizes one to open up or to share private information. Indeed, in its current form, the chatbot stores no personal information of any kind – beyond basic language processing needs, after which it is not currently coded to preserve the user input – nor does it prompt the user to offer any information of that kind. As such, security concerns will not be discussed further in this paper, as their impact on this project is tangential, at best.

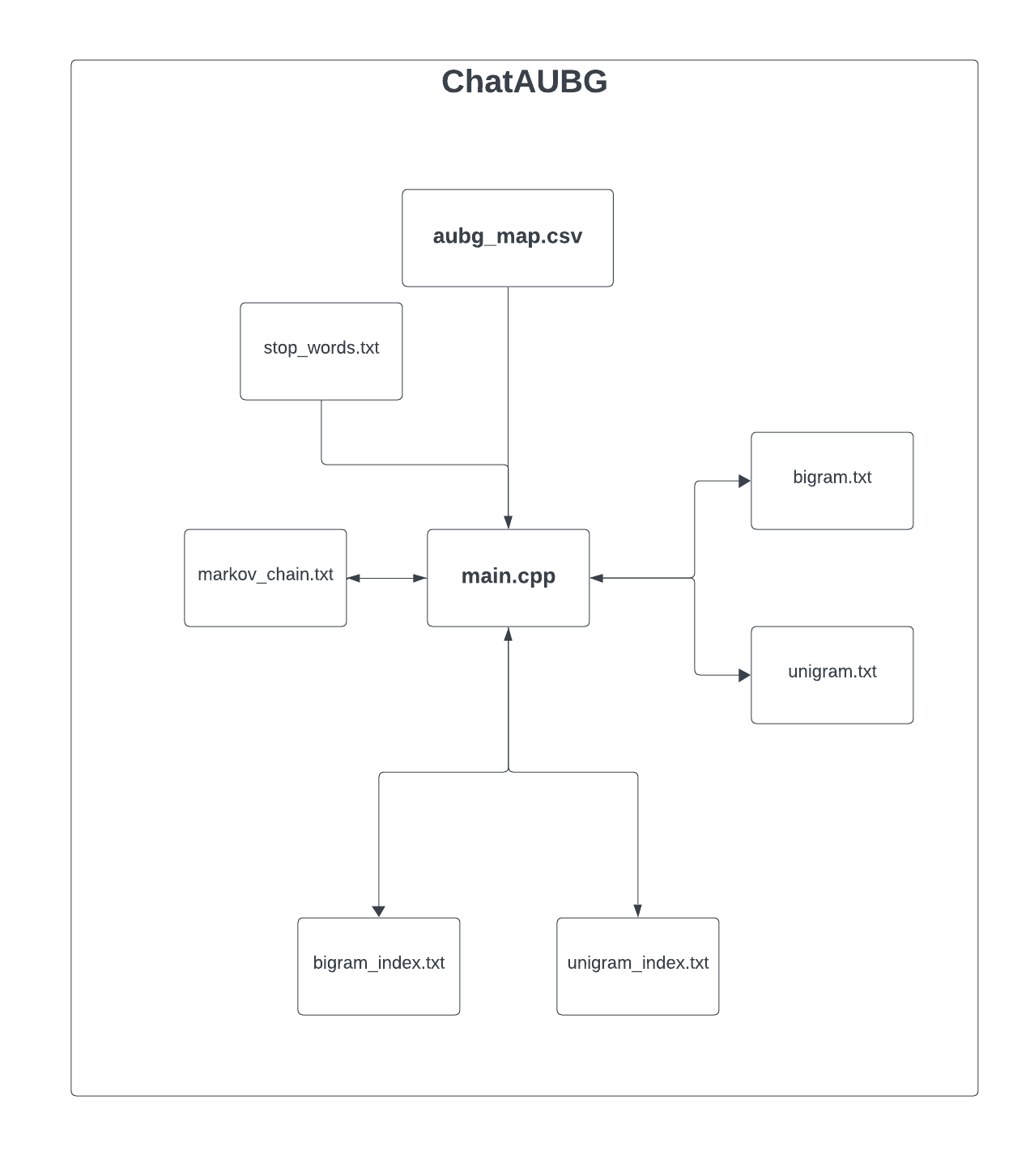
**4. Implementation**

For implementing ChatAUBG, I chose to use C++. C++ is a particularly flexible and powerful programming language [7], and I chose to code in it for those exact reasons: my application falls squarely in the field of Natural Language Processing (NLP), and there is no better support offered by either Java or C# standard libraries. In the absence of particularly convenient languages like Python, I chose to go with the one that offered me the most flexibility, and that imposed the least stringent limitations. Additionally, given the low-level nature of this project, and its overall complexity, I saw little benefit in the additional functionalities of the other programming languages, and so I decided that I would prefer the overall efficiency and speed of C++ [7].

Furthermore, while C++ was built to support object-oriented programming [7], my application does not make extensive use of classes, and so using Java or C# – which are inherently object-oriented languages – seemed unnecessary for this project. C++ does not force me to adopt any particular class-based functionalities, and so it offers me even further control over the abstraction and encapsulation of my project’s functions.

Finally, besides the aforementioned – objective – benefits, C++ is also the language I have the most experience with, and the one I feel most comfortable coding in. Given the relatively high complexity of the task at hand, I decided that I should minimize the degree to which I need to review specific syntax particularities, and instead maximize the focus I can direct towards understanding and developing the specific programming techniques needed for creating ChatAUBG.

In terms of additional libraries used, the entire project was essentially built by only making use of standard C++ libraries. It utilizes the <iostream> library for user input – the code could be further extended to include a more complex UI, but that is beyond the scope of the project at the current time; it uses libraries like <vector>, <unordered\_map>, <string> and <sstring> for their incredibly versatile data structures; it uses <fstream> in order to store and load data to and from external files; it uses the <algorithms> library for accessing functions, for easier, more effective processing tasks; finally, it makes use of the <random> and <chrono> libraries to access pseudo-random number generators, which are utilized for the stochastic textual generation component of the Answer Extraction part of the project.



*Figure 9. UML component diagram illustrating the file structure of ChatAUBG, with arrows indicating the permitted flow of data – e.g. read/write operations being performed as opposed to read only.*

Constructing the initial training dataset is a process that was necessary, and which fundamentally underlines the integrity and effectiveness of ChatAUBG. Thus, I will only now elaborate on my process of creating a proper base of text on which to train my unigram, bigram and customized Markov chain models on.

In order to train my unigram and bigram models – elaborated on in 3. Design of Software Solution – I needed a training set, to compute the probabilities of each word – or pair of words, respectively – appearing on any of the AUBG webpages. I initially considered creating a scraping algorithm, to automatically extract written data from the webpages, and categorize it accordingly for me. I spent some time researching modern scraping techniques [8], but ultimately decided that developing this tool would take too much time, and would be detrimental to developing my main program. I then attempted to solve my problem using manual scraping – i.e., manually copying the words on the webpages, and manually categorizing them. To aid in this process, I used Google Chrome’s “Reading Mode,” that isolates the text form the additional images and hyperlinks, and would allow me to more easily extract it and use it as a training set.

However, as soon as I checked out the sitemap the AUBG website ([www.aubg.edu/sitemap.xml](http://www.aubg.edu/sitemap.xml)), I realized that collecting all of the text data I would need for creating an appropriate training dataset would take too long, and manually copying it would be not only extremely time consuming, but also extremely prone to errors: as I was manually copying and pasting text data, in the infancy of my project, I realized that Chrome’s “Reading Mode” can only scan what is currently on the webpage, so buttons – i.e. “Read More” or “Load More” prompts – posed a problem I did not see any elegant solution to.

As a compromise, I decided to use Web Graph’s “Web Scraper” [9] Chrome extension and create a basic scraping algorithm. This way, I was able to automate the scraping task I needed to complete, while avoiding writing extraneous amounts of code that – per AUBG’s 2024 Senior Project regulations – would not count towards my graded project code. Consequently, this extension allowed me to come up with some very elegant solutions to my problems: I used the aubg.edu sitemap to make sure I cover every webpage I would be interested in; I was able to deal with buttons using Web Scraper’s customizable tools; and I was able to export all of the data I scraped into an XML file, which I could then use to construct my training dataset.

After completing the scraping process, I was left with approximately 17,000 lines of text extracted from somewhere under 400 webpages listed on the aubg.edu sitemap. This number included a slew of duplicates, empty values and non-English text (such as page listings in Bulgarian). I further worked on this dataset in Microsoft Excel, employing in-built Excel functions to create a more manageable and usable dataset. I started by first excluding entries that contained empty text values, then excluding non-English text, and finally manually deleting the duplicate text values; this last process was impossible to automate, because of my scraping method. There were some duplicate values I wanted to keep – such as multiple mention of the word “course” under webpages dealing with information about majors and minors, which I extracted as separate lines of text – and others which were true duplicates that needed to be removed. The only way I could think to resolve this issue was by manually completing this step.

This processing left me with a little over 11,000 lines of text, from a total of 324 webpages. As such, my bigram and unigram classification models would have 324 classes to deal with. I exported my data as a CSV file, and implemented a *void* function declared as *get\_dataframe*, which took as parameters a string consisting of the filename of the CSV file and an appropriate storage structure, to process and store the data in program memory. For that purpose, I used a hashmap – implemented via the standard C++ *unordered\_map<string, vector<string>>* data type – called *website\_map*.

I connected the text extracted from each of the webpages to a key consisting of the webpage’s URL: for example, text relating to contact details – available at https://www.aubg.edu/contact-us/ – would be associated with the string “https://www.aubg.edu/contact-us/.” Naming the map keys in accordance with the URL of the particular webpages should be convenient for keeping track of the origin of every piece of data, and would allow the model to be easily updated as new webpages are added. Furthermore, the storage of website data in a string vector, as opposed to a single large string, would allow for easier modification of the program as webpages are altered – for example, instead or remaking the entire dataset, and retraining the models completely, one could write a function to only change vector entries corresponding to changed text, remove those corresponding to deleted text, and simply add new text as a new vector entry.

Then, as part of the training of each of model, the text data I extracted off of the webpages would be fed through the *to\_lowercare* and through the *smoothe* functions I created – further described functionally in 3. Design of Software Solution. I created these functions mainly for manipulating and modifying the user input that the chatbot receives; however, dynamically altering the content of the training set as we have need to do so, creates a more decoupled environment, where the training set remains altered to use at a later date – perhaps as the webpages get updated – without affecting the proper functioning of the application.

In addition to those two functions, I also created a tokenizer – similarly elaborated on in 3. Design of Software Solution – to “cut up” strings consisting of sentences, and split them up into words. Thus, when processing the dataset created, the textual information would first be smoothened, then converted into lowercase, and finally separated into individual word tokens, stored in a *vector<string>* variable returned by the *tokenize* function.

The Search of Documents function of ChatAUBG relies first and foremost upon a text classification algorithm – further elaborated under 3. Design of Software Solution; as such, the probability that any one word – or pair of words, respectively – to appear in any one webpage must be computed and subsequently stored. For convenient access and retrieval purposes, I created a *Probabilities* struct that holds the particular word or pair of words in a string variable. It contains the following important attributes:

* *string entry\_text* – containing the word, or pair of words, respectively, that the probabilities pertain to.
* *unordered\_map <string, double> entry\_count –* a hash map that maps onto every key of the *website\_map* hash map the particular probability of the *pair* string appearing under that webpage.

As mentioned in 3. Design of Software Solution, Laplace smoothing is an important part of any properly designed Naïve Bayes algorithm. Thus, I initialized every *Probabilities* entry’s *entry\_count* map using the value of 0.0001, which, through testing, I noticed to be small enough for probabilities to still be computed correctly.

For this purpose, I created a *void* method called *build\_map()*,in which I iterate through every key in the *website\_map* hash map, initializing every corresponding entry in *entry\_count* with a value of 0.0001. The entire implementation of this structure can be seen in Figure 10.

I decided to use a *struct* type variable for the implementation of *Probabilities*, instead of a *class* type variable, for two major reasons: firstly, classes are most commonly used for creating object-type associations between data. In my case, *Probabilities* variables are needed for simple associative storage purposes, and do not correspond to actual objects. An object-type implementation would be less appropriate for the purposes I am using it for. Furthermore, because of the low level of abstraction, and the highly mathematical nature of my program, a class would hint at different usage scenarios than it is intended for.

Secondly, classes are private by default, while structs are public by default; in my case, I saw little need for encapsulation. The purpose of encapsulation is to protect certain methods and attributes from wanton modification, as they are being used and reused. However, in my case, Probabilities variables are simple enough not to share this vulnerability. In fact, because of the processing-intensive nature of my program, removing the need for getters and setters will serve to strip off unnecessary components, increasing the program’s speed of execution and lowering the storage space necessary for it. This will serve the program better than a higher level of abstraction and encapsulation would.

I then used the aforementioned structure to create two similar functions for training my classification models. I declared two *void* type functions, named *train\_unigram\_model()* and *train\_bigram\_model()* respectively. Their implementation is fairly similar, so I will describe the process for both at once, noting the differences where appropriate. The specific flow of training is illustrated in the flowchart depicted in Figure 11.

The function first makes use of the *smoothe, to\_lowercase* and *tokenize* functions to create a list of words – or pair of words, respectively – to create a classification model for. It stores the resulting string vector – a vector of words – in a variable declared as *content*. The program checks to make sure *content* is not empty – to account for cases in which we might accidentally scrape a string of words that is either not in English, or not a nonsense string like “?????,???” that might result from errors in converting special characters, where the smoothening function might remove every character in the string altogether – after which it proceeds to iterate through the vector, processing each word in the original string independently.

In the case of the bigram model, instead of processing each word independently, the program processes the words in 2-by-2 pairs. Thus, the condition for checking whether *content* is empty is altered as well, checking instead whether it contains 2 or fewer words – if it does, we cannot group the words in 2-by-2 probabilities, and so we just skip over the string, instead.

The functions also calculate the amount of total words in each webpage as they execute, storing the results in an *unordered\_map<string, int>* variable called *website\_length*. I considered creating a separate function that would iterate through the *website\_map* hash map independently, and calculate these values; however, I realized that the bigram and unigram model cared about completely different numbers. In the case of the unigram model, it was important to consider the number of individual words on the webpage. In the case of the bigram model, however, the program had to consider the number of pairs on the webpage. Thus, these values would be fundamentally different, and calculating them dynamically, as the models are being trained seemed like the most effective and efficient approach.

The functions would then instruct the program to check whether we have already encountered the word or pair of words before – as would be stored in the *entry\_text* field of a *Probabilities*entry; if it had encountered it before, the program merely needs to increment the value of the associated *entry\_count* value, corresponding to the key consisting of the webpage it is currently working on. If it had not encountered it before, it would need to create a new *Probabilities* variable, assign its *entry\_text* field to the word or pair of words that are currently being processed and increment it’s appropriate *entry\_count* value by 1. Afterwards, this value would be added to a vector via the in-build *push\_back()* method.

*A computer screen shot of a program code

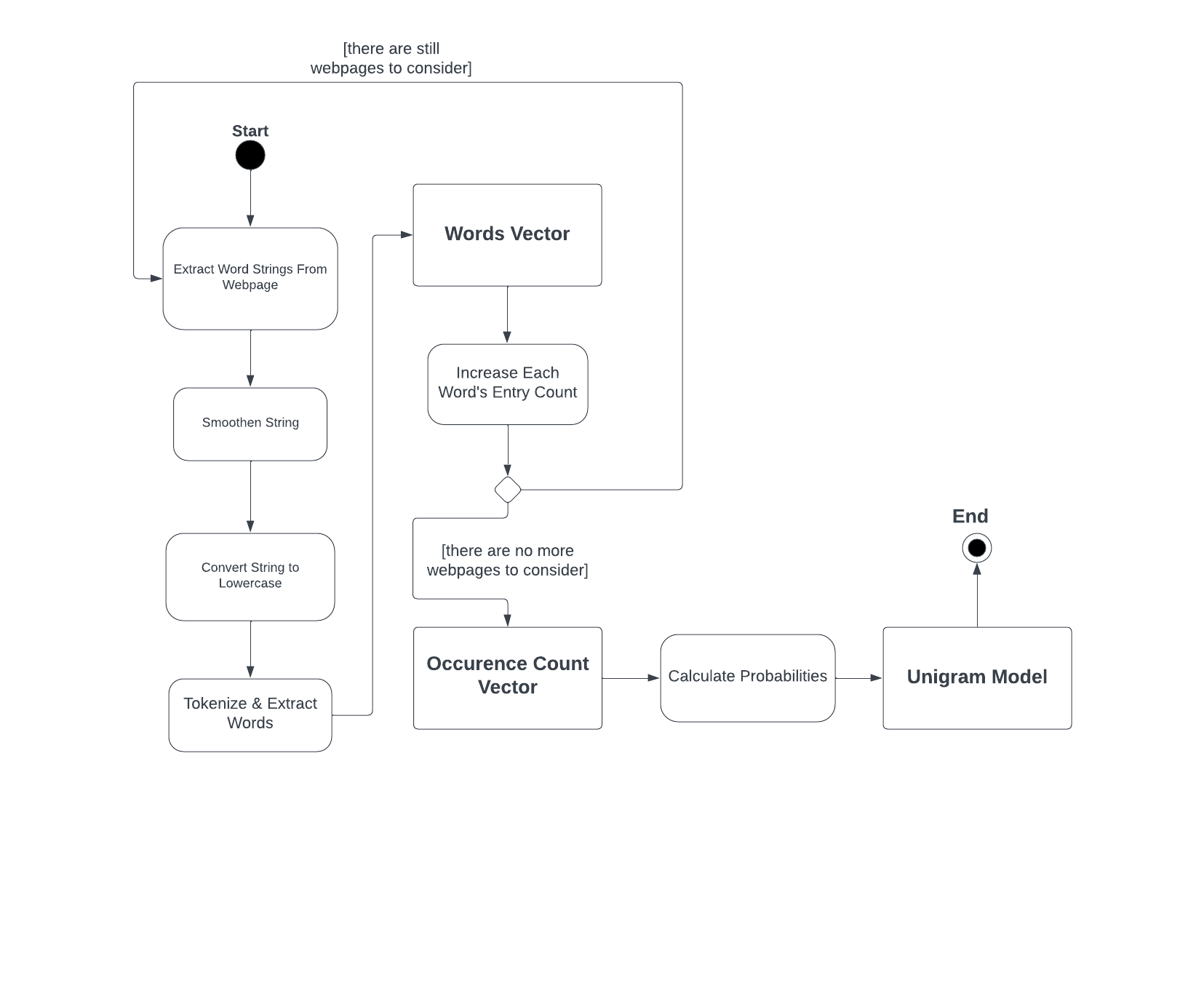
Description automatically generated*

*Figure 10. C++ code snippet, depicting the implementation of the Probabilities struct. Besides the attributes it contains, it also depicts the declaration of two vector variables, bigram and unigram, representing the bigram and unigram probabilities of every word or pair of words in the training set.*

For these purposes, I declared two *vector<Probabilities>* variables, named *unigram* and *bigram*, to store each resulting list of associated texts and probabilities.

The final component necessary for training these classification models was a function that would convert the number of entries counted up into formal probabilities. For that, I created the *calculate\_probabilities()* function, declared as type *void*, which would take as parameter a vector of *Probabilities****.*** It would then iterate throughout each entry’s *entry\_count* map, dividing each value by the amount of words (or pair of words, respectively, as computed by the training function) on each webpage, as stored in the *website\_length* map. This function would simply modify each value in the parameter vector, instead of returning any values.

Finally, in order to minimize the execution times, created a *void* type function that would train every model (including the Markov Chain generative model, elaborated upon later in this chapter), and output the resulting probabilities into a text file. As my dataset is extremely large, training the models can take a high amount of time. In fact, during the many times I have trained and re-trained the classification models, this process took anywhere from 25 minutes to 75 minutes. Thus, it is highly impractical and inadvisable to train each model as we are presenting the program to a user. Instead, I opted to store the resulting values in text files – “unigram.txt” and “bigram.txt” respectively – and load them upon executing the program.



*Figure 11. UML action state diagram depicting the flow of the unigram model training function – void train\_unigram\_model(). The program iterates through the stored webpage text strings, processing them and then splitting them into word vectors. It then counts the number of times each word appears in the webpage text data, compiling an occurrence count vector. Finally, through the calculate\_probabilities() function, it completes the creation of an unigram probabilistic model.*

Finally, I created two functions – one tailored to the unigram model and one tailored to the bigram model – for predicting the website where the information the user is looking for is located in, called *unigram\_predict\_website* and *bigram\_predict\_website*. The method I used for implementing this essentially follows the standard statistical methods for artificial intelligence programs that use Naïve Bayes statistical models: the probability of each word – or pair of words – in the input string appearing in every class is multiplied together to obtain the probability of the entire string appearing in a particular class (which is the key in *entry\_count* with the value attached to it).

As the program deals with very small values, there is the added concern of rounding errors. As such, I modified the normal formula to work with logarithmic values instead of very small subunit probabilities, as is very often done with similar Natural Language Processing tasks [11]. In consequence, the algorithm can no longer multiply values, and instead adds them up; the condition for deciding which webpage to predict would remain the same, however – the key corresponding to the greatest probability would be selected.

After the implementation and training of my classification models, I set out to create a generative text model, that would fulfil the requirements of the third component of my project, that of Answer Extraction. In order to implement this, I created a *struct* called *MarkovChain*, that is comprised of the following attributes:

* *string word* – storing the word corresponding to a particular state in the Markov Chain, to which the transition probabilities are attributed.
* *unordered\_map<string, double> transitions*– a hash map that stores the probability of transitioning to any other word in the dataset, each of them corresponding to a different discrete state in the Markov Chain.
* *int total\_entries* – a value that stores the amount of times a particular word is encountered; this is necessary for calculating probabilities when training the generative model.

Similarly to the *Probabilities* structure, I decided that my purposes would be better served by a struct, rather than a class. Again, this is closer to a connected series of variables, rather than an actual object we can abstract. In order not to hint at usage scenarios for which *MarkovChain* variables would not be conceptually suited for, I decided that a struct would be more appropriate to implement. The full implementation of this is illustrated in Figure 12.

The process of training the Markov Chain model starts similarly to the process pertaining to the training the other two models I previously covered: the program must iterate throughout the *website\_map* hash map and extract particular word strings, which must them be tokenized to extract individual words, stored in a string vector called *content.*

However, my normal preparation functions – *smoothe* and *to\_lowercase,* further elaborated on in 3. Design of Software Solution – would not be a good fit for this particular model. In order to comply with the goals previously stated in 2. Specification of Software Requirements, the chatbot must phrase replies in a manner that approaches natural language, so proper punctuation and capitalization would be highly desirable.

For that purpose, I modified my initial *smoothe* function, adding an optional Boolean parameter to it, pertaining to a *special* type of processing, that would retain commas, dashes, at symbols, exclamation marks, question marks and apostrophes. This way, as the model constructs its transition probabilities, it will be able to reproduce properly punctuated words from the dataset. Similarly, I decided not to use the *to\_lowercase* function, in order to preserve capitalized words.

Similarly to the bigram model, the program must check that, after tokenizing, the *content* vector contains more than one word, or there is no transition value to be gained out of the string, and the program must continue and extract the following string. I then declared a pointer variable pointing to a *MarkovChain* type variable: this is used when deciding whether the word we are currently working on has been previously encountered or not. The construction of the *transition* map is more complicated than that of the *entry\_count* map; therefore, we cannot always just increment a value when we realize the word has been previously found, and further processing is necessary – for example, we might have to add a new key to the map altogether, if the word had not been followed by the particular word that follows it now, before. Thus, this pointer variable will allow us to operate either on an existing *MarkovChain* entry, or on an empty one we can subsequently store.

Therefore, after deciding whether this particular word has been encountered before or not, the program either assigns its value to a new *MarkovChain* variable and stores its address into the pointer, or assigns it the address of where the word has been previously stored, allowing us to operate on that entry.

Next, we either increment the second value of the *transition* entry corresponding to the next word in the string, or we add a new key corresponding to that word and initialize it with 1 – in the case that it did not exist previously. Here, any sort of Laplacian smoothing is not only not necessary, but also counterintuitive: offering the program a small possibility of transitioning to any word in the database would create chaotic replies that would quickly get out of control, and offer nonsense replies.

Finally, as with the other models, I stored the resulting storage vector – declared as *vector<MarkovChain> markov\_chain* – into a text file called “markov\_chain.txt”, so it can be simply loaded in the future, instead of having to be recompiled every time. The final file was approximately 1.4MB large, thus offering no concerns in terms of storage or loading time. The general flow of the function can be observed in the action state diagram in Figure 13.

A screen shot of a computer program

Description automatically generated

*Figure 12. C++ code snippet detailing the declaration of the MarkovChain structure. It contains three attributes, referring to a particular word, it’s associated transition matrix and the total amount of times that word is encountered in the dataset. Additionally, the declaration of the vector used for training and using the Markov Chain generative model is included in the code snippet.*

Finally, I created a function responsible with generating text replies, and one function responsible for the biasing of the Markov Chain model, in accordance with the results of the classification algorithm. Declared as *void bias\_generation*, the function takes as parameters a string variable – corresponding to the appropriate website key determined by the classification algorithm in the second part – and a vector of *MarkovChain* type variables, corresponding to the stored Markov Chain generative model. The function would then construct a new, smaller Markov Chain generative model – stored in another *MarkovChain* values vector called *new\_model* – focused only on the particular webpage selected. The training method is nearly identical to the one used previously for training our main Markov Chain model. At the end of this function’s execution, the transition probabilities of the *MarkovChain* variables pertaining to the words contained in the new model are average out with the ones stored in the original generative model, thus biasing the algorithm in favor of the transition distribution that it would receive from this webpage alone.

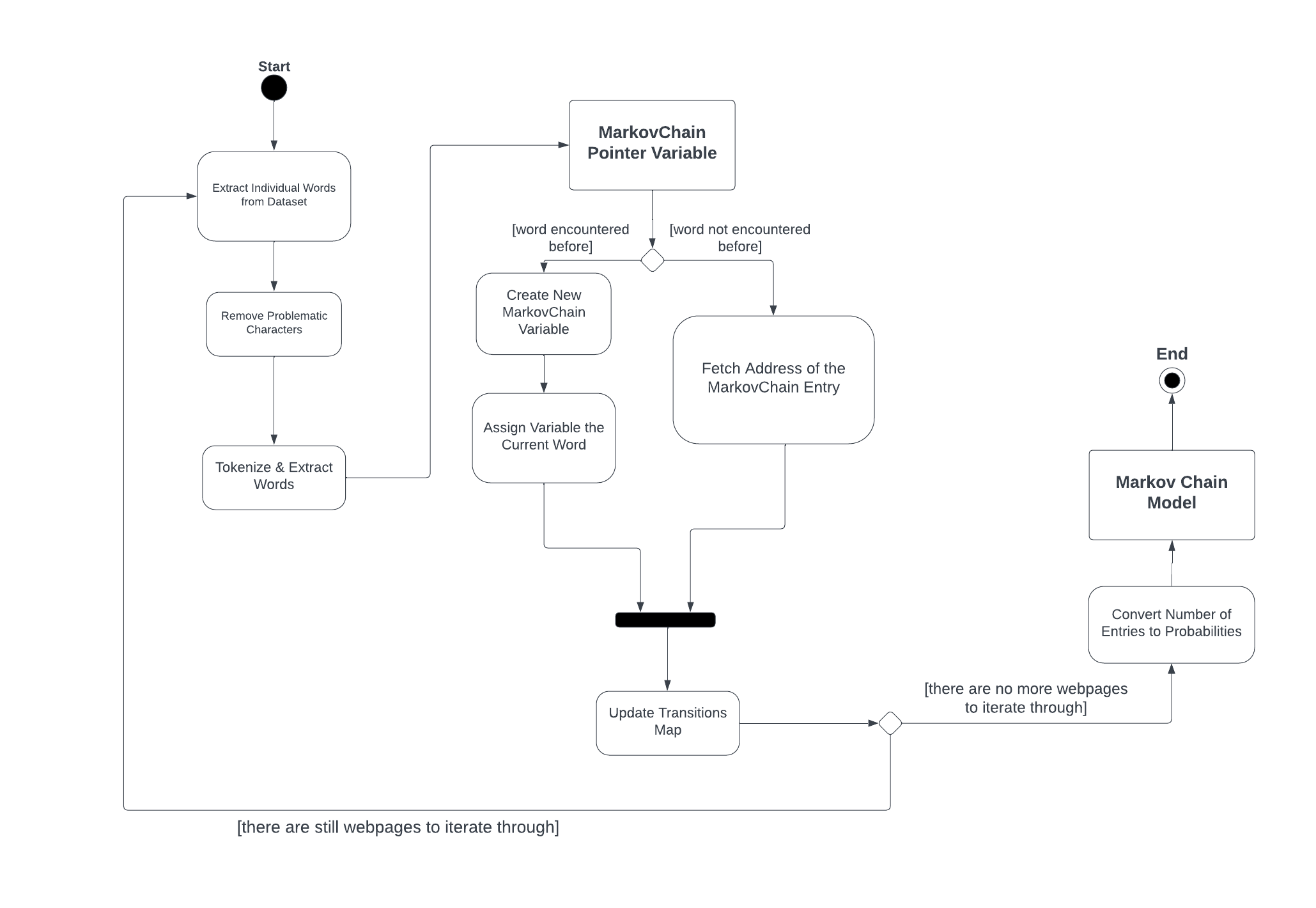
The function tasked with the stochastic generation of textual replies is defined as *void generate\_text*, and takes a parameter the vector of *MarkovChain* values representing the Markov Chain generative model. This function relies on the <random> standard C++ library in order to randomly select how to transition between words, and how to construct a proper reply. Initially, the program selects a random word to start with, and then continues generating text until an ending condition is satisfied. For this generation, the program uses a *default\_random\_number* variable called *rng*, and accesses the <chrono> library to convert the system clock time into an integer – which I called *seed* – which is then used to seed the random number generator.

Using a variable of type *uniform\_real\_distribution<double>*, called *transit*, which can take any value between 0 and 1, the function then selects a word from the *transition* map, by using cumulative probabilities. By the nature of the transition map, the transition possibilities and their probabilities all add up to 1, meaning that one could segment a unit into proportional parts, representing the chance of each word to be selected. Thus, we can use the *transit* variable to select where to transition from any one word, indefinitely, by placing it in the discrete subunit space determined by the word’s transition possibilities.

This generative function can be further observed in Figure 14.

Finally, I considered adding some hard-coded rules to my generative model. As mentioned earlier on in chapter 3. Design of Software Solution, my artificial intelligence model relies mostly on the corpus – or dataset – I built for it. However, we would be remiss not to consider the benefits conferred by giving the chatbot some concrete guidelines for types of words that should carry more weight than others, or for words that should naturally appear at the beginning of a sentence, or at the end of it, respectively.

There were several such rules I implemented: firstly, I modified the *generate\_text* function so that it would always start with a capitalized word in the corpus. This does not always guarantee that the word would fit organically at the beginning of a sentence, but it serves to lower the degree to which replies are hectic and chaotic, nevertheless. Secondly, I modified that same function to only end the answer generation after processing a word that ends with a period – signaling unequivocally the natural end of a sentence. I accounted for abbreviate words by checking that the word in question is at least 3 characters long (so that initials and such abbreviations do not fool the program into thinking that they are sentence-ending structures.



*Figure 13. UML action state diagram depicting the flow of the creation of the Markov Chain generative model. The model tokenizes the strings in the dataset with minimal corrections beforehand, after which it computes the transition map that guides its generative process.*

A screen shot of a computer code

Description automatically generated

*Figure 14. C++ code snippet, illustrating the main generative loop of the program, which continuously selects words based on a Markov Chain-based transition map, until the satisfaction of a particular condition. As can be seen here, the end condition passed here revolves around a minimum reply length, and around the final word ending in a period.*

I also added a function called *validate\_input*, that makes sure that the program is not faced with non-English characters, which would naturally throw the entire model in disarray. Thus, this function checks whether every character in the input string is appropriate, by checking its ASCII value and making sure that is within parameters – i.e. checking that it does not go above the value of 127, into the realm of unsupported special characters.

In the end, I added a function – defined as *void eliminate\_redundancies* – that would handle some often occurring words – such as “who,” “what,” or “where” – that would appear organically in user input, but would not ultimately contribute meaningfully to the execution of the program; instead, they might needlessly confuse the classification algorithm instead. Thus, hard-coding the removal of such words should improve performance in the long run.

**5. Testing**

An additional modification I made to the initial conceptualization of my classification models – the unigram and bigram models, respectively – was trimming the results by relevance. As I was training the models, I noticed that the text files containing the resulting probabilities were extremely large. The initial “bigram.txt” file I computed was 1.7GB large, while the initial “unigram.txt" file occupied 450MB of space. Regardless of the practicalities of storing these files – i.e. the amount of money that storing such files would cost, or the degree to which it would be portable across platforms – this posed an issue to the program itself, in the form of loading time. Regardless of the method I used to load up my models (further elaborated on later in this chapter), this would take a very large amount of time.

As thus, I decided to remove elements from the *bigram* and *unigram* vectors that occurred fewer than two times. The rationale of this is that, in the grand scheme of the program, words that occur once or twice are so rare that their relevance is highly limited. Removing them from the model’s vector would mean that, when encountered in the user’s input, their probability to be classified in either class would be the default one – that is, a static 0.0001 probability. This compromises the accuracy of the model slightly, but in exchange, it drastically reduced the size of the files I was working with. The final version of “bigram.txt” is approximately 374MB large, while “unigram.txt” is approximately 119MB large. This is a significant improvement in space storage, and consequently a significant improvement in the initial loading times of the model.

Another thing I noticed as I was testing these models, was that that I was getting some very strange erroneous results. After some time spent on narrowing down the cause of the issues, I noticed that several webpages in my dataset which contained very few words, some as low as 5 words, would continuously be selected by the program.

Consider the formula I am using to compute the classification probabilities, as implemented in the *calculate\_probabilites* function:

It’s clear to see that, with a high enough difference in the number of words on individual webpages, the number of words on a webpage would exert an undue degree of influence over the final results. In order words, even if no words match any of the words in the class, it might still be selected as the best match, if it contains a low enough number of words.

After this realization, I started modifying my dataset again, manually deleting classes that contained very small amounts of words. In the end, I ended up trimming down my dataset from 324 classes to 250 different classes.

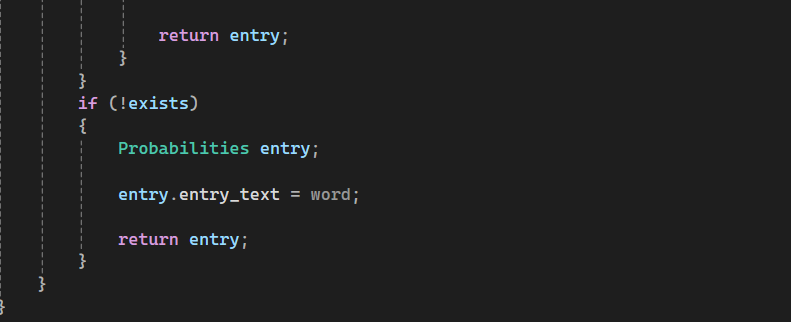
Another important consideration I had to face was the method used for loading in the classification models. Initially, I considered simply reading the files I had used to store the model data into the vectors I had pasted them from originally. While this seemed like a simple solution, it turns out that this method was exceedingly slow. After some testing, the average loading time of the models was between 5 and 10 minutes; this was clearly excessive, and not conducive to a good user experience. Thus, I decided to create an additional loading function, that would only load in parts of the models as they became relevant. I called this function *smart\_load\_model*, and its implementation can be observed in Figures 15 and 16.

The smart loading function *smart\_load\_model* makes use of an additional *void* function called *create\_index*. The latter iterates through both “unigram.txtx" and "bigram.txt” and creates indexes for them, containing the name of every word – or pair of words – previously stores in the *unigram* or *bigram* vectors, respectively, followed by the number of the line at which they are written in the text file. These indexes are similarly stored in text files, at “unigram\_index.txt” and “bigram\_index.txt,” respectively. By making use of these additional files, we can essentially forego pre-loading our classification algorithms in their entirety, instead just loading a fragment of these in a kind of “smart map” – of which I declared two, defined as *unordered\_map<string, int>* called *bigram\_load* and *unigram\_load*.

A screen shot of a computer program

Description automatically generated

*Figure 15. C++ code snippet pertaining to the first half of the smart\_load\_model function. This part deals with the initial part of the loading, where the index file values are loaded into the smart\_map variable, so they can be loaded at a later date, if needed.*

*A computer screen shot of a program code

Description automatically generatedFigure 16. C++ code snippet pertaining to the second part of the smart\_load\_model function. This part illustrates the logic of the program when it needs to load map parts corresponding to words it encounters during its execution.*

Functionally, the *smart\_load\_model* function serves two purposes, and distinguishes between them through an optional string parameter called *word* – which defaults to an empty string. When nothing is passed into the optional string parameter, the function loads the index file – whose name is also passed as a parameter – into the smart map – naturally, also passed as a parameter – performing the initial step needed for this logic to function (code depicted in Figure 15).

When the function receives a non-empty string as a *word* parameter, the function instead looks up the word in the previously loaded smart map, and returns a *Probabilities* type variable, corresponding to the *word* string passed, and fetches its appropriate *entry\_count* map from the original file containing the model probabilities, using the line number stored in the smart maps (code depicted in Figure 16). Naturally, if a string that does not appear in the “bigram.txt” or “unigram.txt” is passed into the *word* string, the function will return a *Probabilities* value that contains the passed *word* string value as its *entry\_text* value, and default *entry\_count* values – as mentioned before, all of the *double* values will be initialized to 0.0001.

In terms of testing and validating my classification models, I noticed a marked difference between trying to classify single meaningful words – like “admissions,” “admission deadlines” and “computer science” – and sentences containing those same meaningful words. For example, “computer science” was safely classified in the “https://www.aubg.edu/academics/bachelor-degrees/computer-science/” class, in unanimous agreement between the unigram, bigram and mixed UBIS model – which seems most appropriate, considering the information available on that webpage. However, a sentence like “Can you give me information about the computer science major?” gave the models significantly more difficulties: the unigram model classified it in “https://www.aubg.edu/academics/bachelor-degrees/physics/”; the bigram model classified it appropriately, in “https://www.aubg.edu/academics/bachelor-degrees/computer-science/”; and most surprisingly, the UBIS model classified it in the “https://www.aubg.edu/academics/bachelor-degrees/literature-minor/” class.

These tests further convinced me of the need for hardcoded rules for weeding out relevant words from filler words: in this example, “can,” “you,” “give” and “me” serve no purpose in terms of classification, and only muddy the waters. Given the limited scope of the chatbot, it is clear that more rules were necessary to weed out as many of these words as possible.

Thus, I started looking into Natural Language Processing ways of removing stop words – which is how they are referred to in professional literature; the unfortunate conclusion is that most approaches are highly complex, and would require at the very least the training of a completely new statistical model to identify the particular words in my dataset which have the appropriate frequency throughout the model, and which are actually confusing my classification models [14]. Sadly, the time constraints of this project would not allow me to implement something like this.

As an alternative solution, I started looking for predefined lists of common English stop words, and found a GitHub repository of an open-source list of stop words, developed using Python’s NLTK library and the tools it provides, and generalized to encompass a large number of natural English stop words [15]. In lieu of a more specialized list, I decided that this list would be an appropriate alternative, that should increase the effectiveness of my classification algorithms nonetheless. I saved this list and included it in the project folder under the name “stop\_words.txt”

After this – rather major – change, I decided to re-train my unigram and bigram models, with the added stipulation of removing stop words from the dataset as I am training them. Theoretically, this should significantly lower the memory storage requirements of the models; according to [14], this improvement is supposed to be around 40% on average. In my case, the size of the “bigram.txt” file was lowered to approximately 181MB – a tremendous improvement – while the size of the “unigram.txt" file was lowered toapproximately 109MB. It seems like implementing a degree of stop words filtering had results that very strongly supported the claims in [14].

The second thing I had to check and stress-test was the generative Markov Chain model. Unfortunately, this was quite chaotic, constructing reply strings with very little sense, despite the additional context offered by the biasing algorithm. It seemed like the random nature of the model was exceedingly pronounced, to a point where it was nearly impossible to understand or extract anything of value from it.

Therefore, another adjustment I made to my project, after its initial conception, was modifying the generative model from a unigram-based one – in the sense that I was only considering the previous one word when creating the transition matrix – into a bigram-based one – i.e., that we are now considering the previous two words when compiling the transition matrix. Fortunately, this only required a few modifications to the training function – *train\_markov\_model* – and after re-training and rechecking the model, the results were already significantly more coherent. This modification did bring with it an increase in the size of the “markov\_chain.txt” source file, up to approximately 2.3MB, but given the small overall size – which still allows me to comfortably load up the entire model on startup, and bias it as the program’s execution continues, I saw this as a definite step in the right direction.

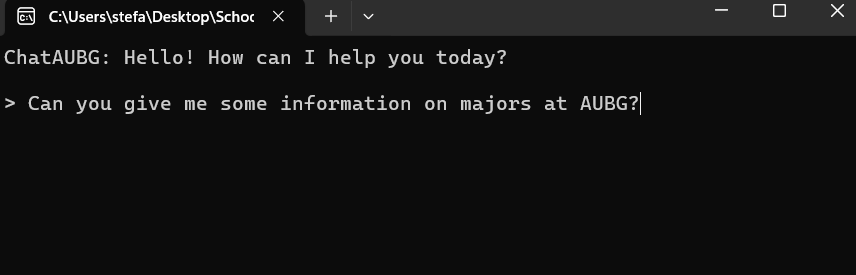
In fact, I was so pleasantly surprised with the results of increasing the level of the Markov Chain for generation, that I decided to increase it one more time. I turned it from a bigram-based model into a trigram-based one – meaning that my program is now looking a full 3 words behind, in order to decide what word to transition to. There are two sides to this change: on the one hand, the generated text is increasingly overfitted – meaning that the generation process shifts more and more from pseudo-random to more and more deterministic; however, I can hardly see how this is an issue currently, considering the output is otherwise extremely chaotic. I would argue, in this case, the data is less so being overfitted, and more so being simply fitted to the requirements of a coherent reply in a better manner.

The final “markov\_chain.txt” file occupies approximately 3MB of memory space, and offers no issues in terms of loading on start-up; the model can be easily loaded up in under a couple of seconds.

**6. Results and Conclusion**

ChatAUBG was almost definitely a learning experience. I set out to build a simple rule-based chatbot, and slowly realized that every single feature I would have to implement was better served by a different approach. Thus, I slowly transitioned to a corpus-based approach, gaining a new level of understanding for the complexities of constructing, working with and operating on a large dataset.

I ultimately did manage to build a piece of Natural Language Processing software; this implements both a complex classification algorithm and a fairly complicated text generation algorithm. I learned about a multitude of Natural Language Processing techniques fairly in depth: I implemented Laplacian smoothing, I took care of stop words, I experimented quite a lot with training and retraining models to optimize for accuracy and storage efficiency, and I learned how much trial and error truly goes into training an artificial intelligence model. I also truly gained respect for the work that goes into curating a dataset – I was involved with mine at every step. I scraped data off the internet, then I had to process it in Excel – which, in hindsight, is probably not the ideal application for doing the kind of processing required for something of this size – and then I still had to spend immense amounts of time attending to it manually, becoming intimately aware of its strong points and its weaknesses.



*Figure 17. A screenshot of ChatAUBG’s initial greeting screen. Currently it is processing the user’s query.*

Indeed, a great deal of both the successes and failures of my project come down to the quality of my dataset. On one hand, it was large enough to have a wide set of possible contexts that my generative model could draw upon, and tailor the possible range of replies to a wide variety of situations. The classification models, with the addition of stop words checks, started performing correctly and funneling user queries into the appropriate contexts.

Where the project is still lacking, however, is in its text generation model. Despite my continuous attempts to improve further with more and more complex generation rules, the replies offered by the chatbot are still severely lacking. I have managed to get them to a point where they are mostly coherent, but they still fail to grasp exactly what the user is getting at, and unfortunately my attempt to offer increasingly accurate and meaningful context for the Markov Chain model failed to get the results I wanted. Unfortunately, the Answer Extraction part of the chatbot requires more components than I have been able to implement into it so far.

Now that I have touched on the main successes and fails of ChatAUBG, I would like to also mention what fixes could be implemented for them. For better textual generation – which currently seems to be the most pressing issue, by far – the chatbot would benefit tremendously from some sort of Named Entity Recognition (NER) functionality. In general, this functionality involves recognizing important details from the user’s input, and potentially focusing on that, instead of trying to isolate some relevant information from a “bag-of-words” as my approach has been [4]. Unfortunately, there is very little one could do in terms of effective Named Entity Recognition algorithms, without relying at least to some degree on previously developed transformers, or language processing models models of some kind, so it quickly became apparent that implementing something along those lines during my project development would be impossible, given the time and software constraints.

However, some NER functionality would greatly compliment my context-generating classification algorithms. One big issue that the generative model has is selecting a proper starting point. In my approach, I simply selected a capitalized word, but with a properly developed NER algorithm, the chatbot could instead isolate and seek relevant terms – like “deadlines” or “major” or “computer science” – through the context provided, and focus its replies around the sentences that contain that term.

Initially, I imagined the difference between NER and context generation would be quite small, so I simply went with a context identification algorithm (via the UBIS classification model, elaborated upon more in chapters 3. Design of Software Solution and 4. Implementation). However, the final result is quite clearly not the same, but it is unclear to me whether further modifications to the rules of the generative model would help stabilize it – at which point the context generated would be able to guide it towards some meaningful replies – or whether the context generated is simply not explicit enough to guide the chatbot into a concrete direction.

However, the failures we are seeing could also be results of an improperly curated dataset. As I have continuously stated, I opted in favor of a larger dataset to enable this corpus-based approach to work to any degree. However, a downside of this approach is that I had way less control over what actually ended up remaining inside the dataset. Approaches in data mining constantly focus around ways of obtaining data better suited for processing [8, 14], and one could likely make an entire project focused on this topic alone. In future development of this application, it would be highly advisable to work on curating a better dataset, perhaps starting with a cleaner web scrape of the website, paying closer attention to the format in which the data is being extracted and to the issues it will raise in the future.

My project was highly focused on backend development, and I did not have the time to develop a more pleasant User Interface – my entire project essentially runs in the command prompt. I would very much like to port the models developed onto a proper UI; this project was designed with the user experience in mind, after all.

Finally, I would conclude that the project has been a limited success. The generative part of the chatbot has sadly fallen short of the goals I set out, but the rest of the components behave within relatively successful margins. The classification algorithms have passed my acceptance tests, after I introduced mechanisms for limiting stop words; my Question Analysis section behaves correctly as well; and the dataset I collected, while imperfect, has actually exceeded my expectations, in terms of what it has enabled me to do. The Answer Extraction is definitely lagging behind, but overall, I would call ChatAUBG a relatively successful project. If nothing else, it serves as an appropriate proof-of-concept of a very basic, corpus-based chatbot, that relies on context and Markov Chains for generating replies.

**7. References**

[1] S. Mohamad Suhaili, N. Salim, and M. N. Jambli, “Service chatbots: A systematic review,” *Expert Systems with Applications*, vol. 184, p. 115461, Dec. 2021, doi: <https://doi.org/10.1016/j.eswa.2021.115461>.

[2] M. Desai, “Paving the way to a Chatbot Revolution?,” *Medium*, Aug. 26, 2019. <https://blog.searce.com/paving-the-way-to-a-chatbot-revolution-c7ea21fa2f2f> (accessed Mar. 05, 2024).

[3] A. Bouziane, D. Bouchiha, N. Doumi, and M. Malki, “Question Answering Systems: Survey and Trends,” *Procedia Computer Science*, vol. 73, pp. 366–375, 2015, doi: <https://doi.org/10.1016/j.procs.2015.12.005>.

[4] N. Akma, M. Hafiz, A. Zainal, M. Fairuz, and Z. Adnan, “Review of Chatbots Design Techniques,” *International Journal of Computer Applications*, vol. 181, no. 8, pp. 7–10, Aug. 2018, doi: <https://doi.org/10.5120/ijca2018917606>.

[5] I. Nikkarinen, T. Pimentel, D. Damián, E. Blasi, and R. Cotterell, “Modeling the Unigram Distribution,” Jun. 2021. Accessed: Mar. 13, 2024. [Online]. Available: <https://arxiv.org/pdf/2106.02289.pdf>

[6] M. Garg, “UBIS: Unigram Bigram Importance Score for Feature Selection from Short Text,” *Expert Systems with Applications*, vol. 195, p. 116563, Jun. 2022, doi: <https://doi.org/10.1016/j.eswa.2022.116563>.

[7] B. Stroustrup, “An overview of C++,” *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, 1986, doi: <https://doi.org/10.1145/323779.323736>.

[8] D. Glez-Peña, A. Lourenço, H. López-Fernández, M. Reboiro-Jato, and F. Fdez-Riverola, “Web scraping technologies in an API world,” *Briefings in Bioinformatics*, vol. 15, no. 5, pp. 788–797, Apr. 2013, doi: <https://doi.org/10.1093/bib/bbt026>.

[9] Web Graph, “Web Scraper - The #1 web scraping extension,” *Webscraper.io*, 2019. <https://webscraper.io/>

[10] M. Bates, “Models of natural language understanding.,” *Proceedings of the National Academy of Sciences*, vol. 92, no. 22, pp. 9977–9982, Oct. 1995, doi: <https://doi.org/10.1073/pnas.92.22.9977>.

[11] A. P. Noto and S. Saputro, “Classification data mining with Laplacian Smoothing on Naïve Bayes method,” *Nucleation and Atmospheric Aerosols*, Jan. 2022, doi: <https://doi.org/10.1063/5.0116519>.

[12] T. Almutiri and F. Nadeem, “Markov Models Applications in Natural Language Processing: A Survey,” *International Journal of Information Technology and Computer Science*, vol. 14, no. 2, pp. 1–16, Apr. 2022, doi: <https://doi.org/10.5815/ijitcs.2022.02.01>.

[13] J. Weizenbaum, “ELIZA---a computer program for the study of natural language communication between man and machine,” *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, Jan. 1966, doi: <https://doi.org/10.1145/365153.365168>.

[14] D. J. Ladani and N. P. Desai, “Stopword Identification and Removal Techniques on TC and IR applications: A Survey,” *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, Mar. 2020, doi: <https://doi.org/10.1109/icaccs48705.2020.9074166>.

[15] Sebleier, “NLTK’s list of english stopwords,” Gist, Aug. 27, 2010. https://gist.github.com/sebleier/554280