BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION APPLIED COMPUTATIONAL INTELLIGENCE

DISSERTATION THESIS

Question Answering System

Supervisor

Lect. PhD. Lupea Mihaiela - Ana

**Author**

Lie Alexandru-Stefan

**2024**

Abstract

Nowadays, there are a lot of tools available that can help users find information about any topic they wish. Most well-known tools are Google, Bing, DuckDuckGo, and the most recent tool that aids users when it comes to question answering is ChatGPT. All these tools, except the latter, work by waiting for a query and based on the query, the most relevant websites are provided to the user, after which the user must find the answer to the query from the returned websites. ChatGPT works a bit differently, it takes a query, searches the answer in the relevant websites and returns to the user with the answer of the query. This way of answering questions took the world by surprise since it is a much more efficient method, especially when there is a lot of information and less time to find it.

In this thesis, we will discuss how to create a system that can work in a comparable way to how ChatGPT functions. We propose a tool that will take queries from users, it will find a relevant document whose main topic is the topic of the query, after which it will find the answer to the query. We chose a dataset that contains a lot of topics to demonstrate that this system is not tied to any specific topic and can be implemented to a large scale with open domain. The system can be segmented into three parts: Question classification, Information retrieval and Answer extraction.

These parts were combined to create a robust system that will understand what needs to be returned as an answer (ex. Individual), will find the document that relates the most to the query and will answer the question with the exact response.

We chose to design this system on the English language since it is less complex than Romanian, but most of the approaches may be implemented with little to no substantial alterations. Making a system that operates on Romanian language will be the next stage and it will also support future development of natural language processing technology. By supporting the Romanian language, our system can extend the target audience and will also help the development of tools that can be used for the Romanian language. By introducing support for Romanian language, the system can be adjusted resulting into a more robust and better optimized question answering system.

Contents

[1. Introduction 6](#_Toc168749200)

[1.1. Motivation 6](#_Toc168749201)

[1.2. Objectives 6](#_Toc168749202)

[1.3. Paper structure 7](#_Toc168749203)

[1.4. Original contributions 7](#_Toc168749204)

[2. Theoretical Background 8](#_Toc168749205)

[2.1. Computational Linguistics 8](#_Toc168749206)

[2.2. Natural Language Processing 8](#_Toc168749207)

[2.2.1. Text segmentation 9](#_Toc168749208)

[2.2.2. N-Grams 9](#_Toc168749209)

[2.2.3. Lemmatization and stemming 10](#_Toc168749210)

[2.2.4. Part of speech tagging 11](#_Toc168749211)

[2.2.5. Dependency parsing 13](#_Toc168749212)

[2.2.6. Named entity recognition 13](#_Toc168749213)

[2.3. Keyword extraction 15](#_Toc168749214)

[2.4. Headword 15](#_Toc168749215)

[2.5. Hypernym 15](#_Toc168749216)

[2.6. Support vector machine 15](#_Toc168749217)

[3. Proposed methodology 19](#_Toc168749218)

[3.1. Datasets used 19](#_Toc168749219)

[3.1.1. TREC dataset 19](#_Toc168749220)

[3.1.2. Question answering dataset 19](#_Toc168749221)

[3.2. General description 20](#_Toc168749222)

[3.3. Question classification 21](#_Toc168749223)

[3.3.1. Unigrams and bigrams 22](#_Toc168749224)

[3.3.2. Headword 22](#_Toc168749225)

[3.3.3. Hypernym 22](#_Toc168749226)

[3.3.4. Query expansion 22](#_Toc168749227)

[3.3.5. SVM 22](#_Toc168749228)

[3.4. Information Retrieval 23](#_Toc168749229)

[3.5. Answer Extraction 24](#_Toc168749230)

[3.5.1. Keywords extraction 24](#_Toc168749231)

[3.5.2. Preprocessing 25](#_Toc168749232)

[3.5.3. Relevant sentences and answer extraction 25](#_Toc168749233)

[4. Software Implementation 27](#_Toc168749234)

[4.1. Technologies used 27](#_Toc168749235)

[4.1.1. Python 27](#_Toc168749236)

[4.1.2. Pycharm 27](#_Toc168749237)

[4.1.3. SpaCy 28](#_Toc168749238)

[4.1.4. NLTK and WordNet 29](#_Toc168749239)

[4.1.5. Sklearn 29](#_Toc168749240)

[4.2. Workflow of the system 30](#_Toc168749241)

[4.3. Application diagram 32](#_Toc168749242)

[4.4. SVM training 33](#_Toc168749243)

[4.4.1. Unigrams and bigrams 35](#_Toc168749244)

[4.4.2. Headword extraction 35](#_Toc168749245)

[4.4.3. Hypernym extraction 35](#_Toc168749246)

[4.4.4. Query expansion 37](#_Toc168749247)

[4.4.5. List of features 38](#_Toc168749248)

[4.4.6. SVM training 39](#_Toc168749249)

[4.5. Graphical interface 41](#_Toc168749250)

[4.5.1. GUI class 41](#_Toc168749251)

[4.6. Question classification 44](#_Toc168749252)

[4.7. Information Retrieval 45](#_Toc168749253)

[4.7.1. Document and question processing 45](#_Toc168749254)

[4.7.2. Vector of frequencies 47](#_Toc168749255)

[4.7.3. Euclidian distance 47](#_Toc168749256)

[4.8. Answer Extraction 49](#_Toc168749257)

[4.8.1. Keyword extraction 49](#_Toc168749258)

[4.8.2. Relevant sentences 51](#_Toc168749259)

[4.8.3. Answer extraction 55](#_Toc168749260)

[5. Results 58](#_Toc168749261)

[5.1. SVM Results 58](#_Toc168749262)

[6. Conclusion and further development 59](#_Toc168749263)

[6.1. Conclusion 59](#_Toc168749264)

[6.2. Further development 59](#_Toc168749265)

[7. Bibliography 60](#_Toc168749266)

# Introduction

## Motivation

In ancient times, knowledge used to be held by the oldest members within a community, they had the duty to learn and satisfy the curiosity of new generations. Over time, more ways of storing information, such as books, were developed. Anyone who could read could document themselves in a variety of fields by going to the nearest library and looking for the right book. With the creation of the WEB, however, anyone who has an Internet connection and owns information related to a particular topic, could upload it online, making it available worldwide. Thus, we are at a time when access to information can be achieved in the easiest viable way in our history as a species.

People today have access to an unimaginable amount of information compared to the past; this information is also a couple of clicks away from accessing it. The main problem is that with massive amounts of information it will also take a lot of time to find the answers. Even if the process itself can take a couple of milliseconds, finding the actual answer to a question can take a lot of time. Creating a tool that will help users find the information in seconds will greatly improve the efficiency and productivity of people by helping them access the necessary information. By retrieving quick and accurate information, tools like this can revolutionize how easy people can access data. This will help all kinds of people from students, professors, medical personnel, or even curious individuals that want to know the answer to a specific question. Saving time is one of the most wanted features today and creating a tool that can do that will be highly desired. For this reason, automated question answering systems were invented, which will take this task from the shoulders of the users. This question answering system provides a correct, concise, and fast answer to a question formulated in natural language.

## Objectives

The main objective of this question answering system is to wait for an input (query) written in natural language by the user and to have a correct answer be returned to the user in the fastest possible. To achieve this objective our system needs to do the following steps:

* Create and train an SVM that can predict the answer needed for a question.
* Retrieve the most relevant document based on the query entered by the user.
* Retrieve an answer that matches the prediction of the SVM from the relevant document and print it.
* Create a tool with GUI that combines all the previous objectives.

## Paper structure

This thesis paper is structured in the following way:

* Chapter 2: Presents the theoretical background of Computational linguistics, Natural language processing, Keyword extraction, Headword, Hypernym and Support vector machines. This chapter is important since it serves as a basis for the topics that will be discussed in the next sections.
* Chapter 3: Presents our proposed methodology to solve the main subject of the thesis, which is question answering. Here, a more theoretical approach of how the problem is solved and is composed of Datasets used, General description of the problem and solution, Question classification, Information retrieval and Answer extraction.
* Chapter 4: Consists of a more practical approach to the problem. In this section Technologies used will be discussed, Workflow of the system and how the theoretical part present in the previous section (Chapter 3) is implemented.
* Chapter 5: Presents the conclusion drawn from working on this project alongside with what could be further developed to make an even better system.
* Chapter 6: Contains that bibliography used to create this project and thesis.

## Original contributions

One original contribution comes from the information retrieval part of the paper, namely how document relevancy is done. A matrix of frequency is done on every preprocessed document and on the query itself. After the matrix is done, using Euclidian distance we can calculate how similar is a document compared to the query.

Another original contribution is how the answer is extracted from the document. After finding the most relevant n sentences from the document, we will use named entity recognition to get all the tokens that are of the same type as the answer needed for the query. These candidates will be scored based on the relation with the keywords and the highest score is the answer. Sometimes mismatches can happen and if so, the first sentence is returned since most of the time it contains the answer.

# Theoretical Background

## Computational Linguistics

The scientific and technical field of computational linguistics [1] studies spoken and written language from a computational point of view and develops tools that process and produce language in a practical way, either in large quantities or during conversation. Because language reflects the mind, a computational knowledge of language also offers insight into intelligence and thought processes. Language-competent computers would also greatly facilitate our interaction with various machines and software and put the vast textual and other resources of the internet at our fingertips in ways that truly meet our needs, since language is our most natural and versatile means of communication.

In computational linguistics, theoretical objectives include developing grammatical and semantic frameworks to characterize languages in a way that makes syntactic and semantic analysis computationally tractable; finding processing strategies and learning principles that take advantage of language's structural and distributional (statistical) properties; and creating computational models of language processing and learning that are both plausible from a cognitive and neuroscientific perspective.

The field has a wide range of practical objectives. Among them are: effective machine translation (MT); text summarization; analysis of texts or spoken language for topic, sentiment, or other psychological attributes; dialogue agents for completing specific tasks (purchases, technical troubleshooting, trip planning, schedule maintenance, medical advising, etc.); and, finally, the creation of computational systems with human-like competency in dialogue, in language acquisition, and in gaining knowledge from text.

## Natural Language Processing

Natural language processing is a branch of the vast domain known as artificial intelligence. Its main focus is on how a computer can interact with human language [2]. More specifically, it addresses how computers are designed to handle and evaluate vast volumes of natural language data. Making a machine capable of comprehending document content, including the subtleties of the language used in context, is the aim of natural language processing. Subsequently, the technology can precisely retrieve the data from the papers and categorize and arrange the documents.

### Text segmentation

The process of dividing a document into smaller sections, known as segments, is known as text segmentation. Text processing makes extensive use of it. Every part has a purpose that is appropriate. Depending on the goal of the text analysis, the segments are classified as words, sentences, topics, phrases, or any other information unit [3].

In most natural language processing systems, the first preprocessing step that the user-provided text will go through is tokenization (or text segmentation) [4]. Its purpose is to convert the source text from a representation in naturally occurring language understandable by humans to one that is easier to use within our system's framework. To do this, it splits the text into multiple distinct parts, or tokens, and removes some components, such as punctuation. A series of characters that have been taken out of a document and put together because they constitute a valuable semantic unit is known as a token.

A collection of delimiters, which often stand for non-alphanumeric characters, is required to separate the text in a way that will yield the most accurate segmentation. The space character, which is the most basic and widely used delimiter in the English language, is perfect for segmentation since it serves the same purpose as a word boundary in natural language. The following are more instances of often used delimiters: quote marks, apostrophe, period, square or round brackets, etc.

Furthermore, since text segmentation is not always necessary when encountering a delimiter, we will also want a set of well-defined rules pertaining to the circumstances in which they will be used. As an example, consider the dot delimiter:

* The text should be divided by it if it appears at the conclusion of the sentence since it serves to separate the sentences inside a paragraph.
* The text should not be split if it appears inside an acronym or a web address since the tokens produced as a result would be meaningless when taken separately.

### N-Grams

Natural language always has a context (a word might imply various things in different phrases or contexts), therefore processing tokens separately can result in the loss of contextual information, which can be crucial in many NLP tasks. To obtain contextual information from the group of processed tokens, neighboring tokens can be processed together using N-grams. In the N-gram, N is the number of nearby tokens being examined simultaneously. In contrast to the bag-of-words model (which is equivalent to a uni-gram, since N = 1), which prevents us from learning the token order (living room and room living would be viewed as the same thing), bi-grams or N-grams with N > 3 allow us to learn this information (living room would not be the same as room living). NLP uses n-grams for a wide range of activities related to natural language processing. N-grams provide a more complex understanding of text and enable more accurate language processing by considering word context. Some of the many applications of N-grams in NLP are:

* Text prediction: Applications such as text generation and autocomplete benefit from the ability to forecast the next word in a sequence by examining the most common n-grams.
* Language modeling: For tasks like auto-completion, speech recognition, and machine translation, N-grams can be used to represent the probability distribution of words in a particular language.
* Information retrieval: Through effective text indexing and searching, N-grams can yield pertinent responses for searches including just partial words.
* Named entities, such as people, locations, organizations, dates, and more, are recognized and categorized by Named Entity Recognition (NER) systems using n-grams.
* Sentiment analysis: A n-gram analysis helps to understand the sentiment expressed in textual content by pointing out words and phrases in their context.
* N-grams are used as features in machine learning models to separate text into predefined categories for text classification.
* Language generation systems and chatbots use n-grams as the foundation to generate coherent and lifelike text.

Other uses of N-grams are that it can be used as features in machine learning model training for tasks like text classification and sentiment analysis [5]. The usage of N-grams as features is present in this paper as well.

### Lemmatization and stemming

Due to grammatical considerations, a term may appear in a document in several derivative forms, depending on the context. For instance, the terms "eating" and "ate," which essentially mean "to eat," are written differently in the sentence "I am eating right now while she ate two hours ago." This presents a challenge for the computer when it comes to question classification systems. Finding out whether two queries have the same meaning would be far more significant than judging them to be different due to a minor grammatical alteration in a crucial term, which would matter to a human but is meaningless to a computer system.

During the preprocessing stage of a document, the most significant words are typically extracted from the text to determine the category to which it belongs, how similar it is to other documents, and so on. These words must then be stored in a data structure within the application. Storing multiple words with the same meaning is inefficient both from a storage space perspective and a program execution speed perspective, as we will have to go through it again later. Two approaches to solving this problem exist: stemming and lemmatization. The process of lemmatizing [6] a word involves taking it back to its most basic dictionary form, or what is known as the term's lemma. While lemmatization determines a word's dictionary form based on the context in which it appears in a sentence (i.e., it's part of speech), stemming is a much simpler process that involves actually cutting a word's prefixes and endings in accordance with a predetermined set of rules, with the hope that most of the time, two derived forms of the same base word will coincide after the stemming process.

It is impossible to declare which of these approaches is superior since how they are used mostly relies on the requirements of the application that was built. The accuracy and execution speed of the two approaches are where they diverge most. As a result, lemmatization will typically provide a more accurate result since it considers the word's context. However, because of this, it processes the working word more simply than stemming, which results in less accuracy and faster processing.

### Part of speech tagging

Since a word's meaning can change depending on the context in which it appears in a sentence, we are trained from an early age to examine sentence components from a morphological point of view in the languages we speak. For instance, we may think about the next two queries:

„Have you finished reading that book I gave you?”

„Did you book our flight?”

*Example 2.1: Different meaning of the same word based on the context*

In this instance, the word "book" has the exact same form in both inquiries, yet it is employed with distinct meanings (noun: book, verb: to reserve). Staying poses a challenge to query categorization systems. We must understand the precise meaning of every pertinent word we process to tackle this dilemma. A tagger, which serves as a morphological analyzer of the phrase and returns the word's part of speech, can be used to tackle this problem. There are two distinct concepts that this tagger may operate on [7]:

#### Rule based assignment

In this kind of part-of-speech assignment, we build a set of rules based on contextual information that allow us to anticipate the part-of-speech of a word that arises in the same context. This contextual data may belong to the word's lexical form and the words surrounding it in the sentence. One kind of rule that this tagger may use is something like this: "We can consider a word as an adjective if it is preceded by an article and followed by a noun." This approach of allocating portions of speech is highly inefficient since each rule needs to be manually generated by a programmer. If we process a significant quantity of data, this will result in a lengthy list of rules [8].

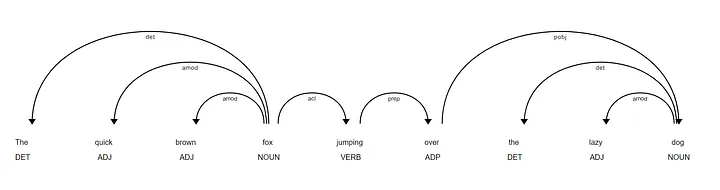
#### Stochastic assignment

This category of assignments includes any assignment approach that determines a word's part of speech using a probabilistic model. Figuring out how frequently a word appears in each portion of speech within a training data set is a very basic initial step in the stochastic assignment process. When training is over, the most often occurring part of speech for a given word will be chosen, and if the term recurs, that part of speech will be assigned immediately. For instance, the tagger will recognize the word "book" as a noun and label it as such each time it appears in the training. The next strategy would be for each word to determine which portion of speech is most likely to follow such a sequence by considering the final N words in the phrase. This method is far more accurate than the one that was previously described since it does not impose the restriction that a word may only have one part of speech; rather, it allows words to vary based on the context, just as it does in real life [9].

Combining the two methods above—using a frequency table of the parts of speech and probability as a part of speech to appear after a string of N words—would be an additional, more sophisticated method. We refer to this method as the Hidden Markov Model [9].

### Dependency parsing

The idea behind dependency parsing [10] is that every word in a sentence has at least one direct relationship with every other word. They are referred to as syntactic dependencies. To effectively extract information from a given text, it is important to examine them to analyze the sentence's grammatical structure [11].



*Figure 2.1: Dependency parsing example*

The diagram presented above explains the structure of dependencies between the words of this sentence. The syntactic dependency between two words is represented by directed arc. The word indicated by the arrow, for example "lazy" is called the component dependent (child) while the word present at the opposite end of the arc, "dog" is the main component (head) of the dependency. Also, on this arc we can also notice the dependency relationship between the two words, namely "amod" which is an abbreviation for "adjectival modifier" which means that "lazy" is an adjective that modifies its meaning the noun "dog".

### Named entity recognition

Exact name entities, such as people, places, and organizations, may be recognized and extracted with significant use when mining text for information. Tackling Named Entity Recognition (NER) challenge involves learning to extract names from natural language text. Most issues in popular research fields including bioinformatics, machine translation, video annotation, information retrieval, question answering and summarization systems, and semantic web search can only be resolved with proper named entity identification and extraction. These days, a growing number of researchers recognize names from text using various techniques such as Rule-base NER, Machine Learning-base NER, and Hybrid NER. Named Entity Recognition (NER) is a subproblem of information extraction that deals with analyzing both structured and unstructured materials and locating references to individuals, locations, businesses, and organizations [12].

The fundamental job of natural language processing (NER) forms the basis of the NLP system. Two tasks are involved in NER: first, proper names in text must be identified; second, these names must be categorized into a set of predefined categories of interest, such as person names, organizations (companies, government agencies, committees, etc.), locations (cities, countries, rivers, etc.), date and time expressions, and locations. In the sixth Message Understanding Conference, the term "Named Entity" was first used (MUC-6). The MUC conferences were the ones that made a significant impact on this field's study. For named entity systems that carried out various information extraction tasks, it served as the benchmark [13].

Since most named items have beginning capital letters and are therefore clearly recognizable, NER is intuitively straightforward for humans, but it is quite difficult for machines. Although most named entities are proper nouns, it is incorrect to assume that the named things can be categorized using dictionaries. There is a constant creation of new proper nouns over time.

Thus, a dictionary could never contain all those appropriate nouns. Determining the senses of named items is a challenging task, notwithstanding their registration in the dictionary. Semantic (or sense) ambiguity accounts for the majority of NER issues; in contrast, a proper noun might have many meanings depending on the context [14]. To provide an example, when is "The White House" a place and when is it an organization? When did someone get the name "June"? What month is it named for, and when? Similarly, although the White House is a place in "He visited Bush at the White House," it is an organization in "White House announced the list of ministry candidate."

For several applications, including machine translation, information retrieval, question answering, and summarization, automatically extracting proper names is helpful. In many situations, the asking point correlates to a NE since the key to a question processor, for example, is identifying the asking point (who, what, when, where etc.). The named entity system in biology text data can automatically extract specified names (such as names for proteins and DNA) from unprocessed texts. The objective of named entity identification and extraction is to identify and categorize names according to their sense from text into certain groups.

## Keyword extraction

Keyword extraction is the process where a list of words is extracted from a text or a document. Keyword extraction is crucial when it comes to NLP. One usage for this list might be as the document's keywords which can be used by search engines to help with their categorization process by using it to filter out papers that only include a few words that match the keywords they are searching for, rather than searching through every page that is accessible for terms. The retrieval procedure will go more quickly and easily as a result.

## Headword

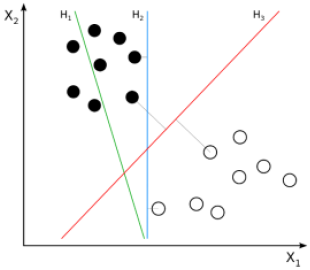
We can define a headword as the word in a question that most clearly signals the kind of information being sought, even if the scientific community has not yet reached a consensus on a definition for this attribute. We may claim that the term "city" serves as its headword in the case of the query, "What is the smallest city in Romania?"

## Hypernym

One phrase that designates the semantic domain that another word belongs to is called a hypernym. Use of the hypernym as the question's headword can lead to generalization without sacrificing information that is essential to the question's classification, as the information that a question seeks might occasionally be quite precise. As an instance, "color" is the hypernym for the word "red".

## Support vector machine

Support vector machine is a supervised learning algorithm used in data analysis for classification or regression analysis. Given a training data set, where each example is classified into one of two categories, a SVM algorithm will build a model that will assign new examples to one of two categories. Each example in the training data will be represented in space as a vector of dimension N. The goal of the algorithm is to find an optimal hyperplane (a subspace whose size is 1 smaller than the size of the space of which it is a part) which best delineates the two categories to which the examples belong. Because there could be multiple hyperplanes to delimit the examples, the algorithm must choose the best one, namely, the hyperplane that has the distance to the nearest point in each maximized category. If such a hyperplane exists, it will be called the "hyperplane of maximum separation". [15]

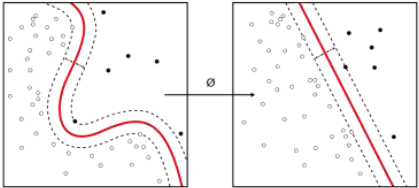


*Figure 2.2: SVM example*

In the example above (*Figure 2.2*), we can observe 3 hyperplanes: h1, h2 and h3:

* Since it is unable to divide the dataset into two groups, the H1 hyperplane is not the best option.
* Although H2 can separate these two groups, there may be too little space between them, which might lead to inaccurate findings.
* The H3 hyperplane divides the dataset into two distinct categories and maximizes the distance between the two nearest locations, making it the best option.

As the data set would not always be linearly confined, it was suggested that every point in the initial N-dimensional space be mapped onto a much bigger domain, presuming that the data set might be more easily constrained there.



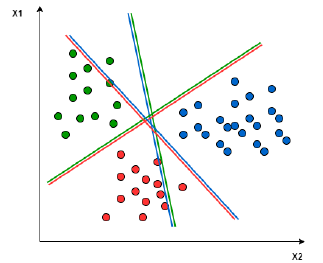
*Figure 2.3: Mapping of the dataset into a bigger domain*

To solve the problem in the fastest method possible and to ease the computational systems, the mapping should be done in an easy manner using the appropriate kernel function for each use case. The most popular kernel functions used are the following:

* Polynomial function: A more generalized representation of a linear kernel. Using this function will not lead to perfect results since the function is not the most efficient.
* Gaussian radial function: One of the most used kernel functions in an SVM, this function is usually used for nonlinear data.
* Sigmoid function: Usually used in neural networks.

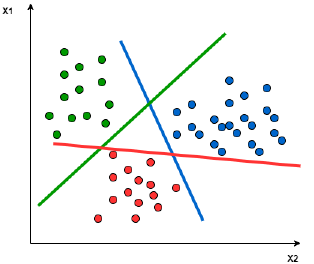
Classification of a dataset that is divided into more classes is a great problem for an SVM since in its initial form, it is equipped to deal with data that can be classified in only two classes. To solve this problem, two solutions were proposed:

* “One-to-one" classification: Proposed the usage of a hyperplane at a time to separate any two classes, neglecting the other classes. To make this happen, the classifier will use N SVM’s, each one of them will predict the class if the element is of his class.



*Figure 2.4: Classes delimitation using “one-to-one” classification*

* “One-to-rest" classification: Proposes the finding of a hyperplane that can separate one class from all others at the same time. This separation will take place in the entire dataset into two categories: points that will be part of the current class and the other ones that will not. To implement this method, the algorithm will use SVM’s.



*Figure 2.5: Classes delimitation using “one-to-rest" classification*

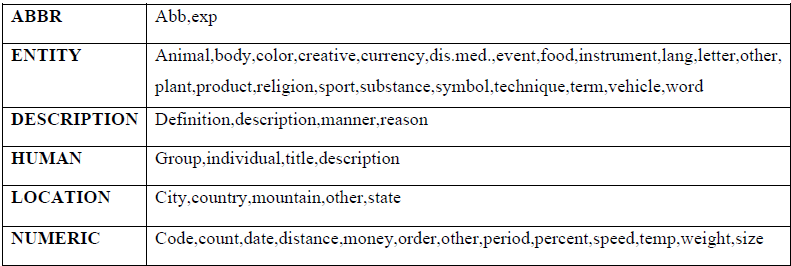
# Proposed methodology

## Datasets used

### TREC dataset

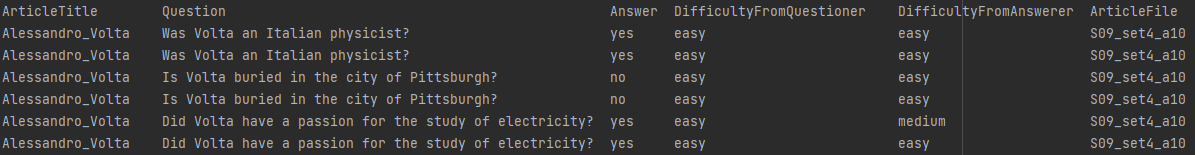
TREC (Text Retrieval Conference) [16,17] is a collection of scientific workshops whose purpose is to further information retrieval by giving researchers access to the tools they need to conduct extensive comparative analyses of various information retrieval techniques. There are 5500 questions in the training set and an additional 500 questions in the test set of the TREC dataset for question classification. Four diverse sources provided the data: 4500 questions from USC, around 500 personally constructed questions for certain unusual question classes, 894 questions from TREC 8 and TREC 9, and an additional 500 questions from TREC 10 conference, which served as the test set.

Within the dataset, questions are 10 words long on average, and there are about 8700 unique words total. These questions were divided into 50 more precise categories that are so detailed that they may be thought of as the expected kind of response for a particular topic, in addition to six broad categories.

 *Figure 3.1: Visual representations of TREC dataset*

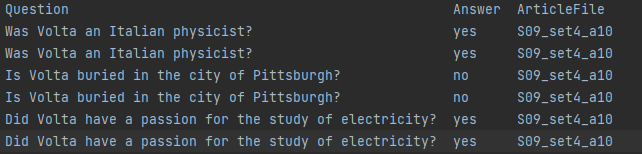
### Question answering dataset

Between 2008 and 2010, several students from Carnegie Mellon University and the University of Pittsburgh, including Noah Smith, Michael Heilman, Rebecca Hwa, Shay Cohen, and Kevin Gimpel, gathered this data [18]. This dataset is used to gather questions in natural language that can be used as input along with the answer and the relevant document where the answer can be found.

*Figure 3.2: Question answering dataset*

In *Figure* *3.2* we can see how the question answering dataset looks like. The dataset is composed of several columns: Article title, Question, Answer, Difficulty from questioner, Difficulty from answer and Article File. We can also observe that all the questions are doubled, this is because the question was used on two different answering systems and the answers may vary. This dataset will be helpful when it comes to the development of the system because it is composed of multiple domain questions, meaning that theoretically our system is not tied to a specific domain and can answer any question that will be fed by the user.

Before using the dataset, some preprocessing needs to be done. Columns that are useless to our system will be removed, namely Article title, Difficulty from the questioner and Difficulty from the answerer. Our system will get a question as input, will find the document that can contain the answer and lastly will return the direct answer or sentence that can contain the answer.

*Figure 3.3: Question answering dataset after preprocessing*

## General description

The main purpose of a question answering system is to give the user a correct answer to the question. This can be achieved using a wide range of techniques from computational linguistics, information retrieval and natural language processing [19,20,21].

Our system will receive a question that can be formulated in a natural language (English). This is already an advantage since a lot of question answering systems receive as input a set of keywords. The system becomes more appealing to users when it accepts questions written in natural language as input, but it also becomes harder to build since additional steps need to be taken. This may cause the system to become sluggish, which may influence many users to select alternative systems that provide faster results. It is important to strike a balance between user appeal and quickness.

Based on this, our question answering system can be categorized into three main sections:

* Question classification module: Whose purpose is to provide what type of answer needs to be extracted from the document and to determine the keywords of the query.
* Information retrieval module: This module will search in a dataset of documents and will return the most relevant document based on the question asked by the user.
* Answer extraction module: Will take the information received from the last two modules and return the answer or the sentence that is most likely to contain the answer to the question.

## Question classification

For our system to return a relevant answer, firstly, it must know what type of answer must be returned. The answer is closely related to the question so we can say that question classification plays a key role in achieving an accurate answering system.

To classify a question, it must first be represented as a vector of characteristics that a machine learning algorithm (like SVM or Naive Bayes) would be given. Therefore, the characteristics that are retrieved from the query should include as much pertinent information about the type as feasible, but they should also be expressed as succinctly as possible to avoid significantly accelerating the program's execution.

To achieve this, we decided to use the following features which contain lexical, syntactic, and semantic information from the original question: Unigrams, Bigrams, Headword, Hypernym and Query expansion.

### Unigrams and bigrams

Are different iterations of the N-gram pattern, which is an uninterrupted string of N phrases extracted from a text. These phrases are even the words that comprise the initial inquiry in our application. An n-gram of dimension 1 will be referred to as a unigram and an n-gram of dimension 2 as a bigram using the Latin number prefixes. We can include the speaking element for each word in the unigram sequence to incorporate more grammatical information.

### Headword

We can define a headword as the word in a question that most clearly signals the kind of information being sought, even if the scientific community has not yet reached a consensus on a definition for this attribute. Using the question "What is the smallest city in Romania?" as an example, we may state that "city" is its headword.

### Hypernym

A hypernym is a phrase used to designate the semantic domain that another word belongs to. Using a hypernym for the question's headword can help generalize the information it seeks without sacrificing information that is essential to the question's classification, as questions might occasionally seek for extremely particular information. As an illustration, "color" is the hypernym for the word "red".

### Query expansion

Query expansion is a similar feature to hypernym that will help the SVM have better accuracy. We take the hypernym and iterate through the syntactic tree level by level. For each hypernym we find, we will assign it a weight. This weight will be smaller the higher we are on the syntactic tree.

### SVM

An SVM will be trained on TREC dataset to accurately predict what type of answer needs to be returned from the query. Once the SVM was trained, a vector of features consisting of unigrams, bigrams, headword, hypernym, and query expansion will be fed to the SVM. The SVM will predict one of the precise categories that can be seen in *Figure 3.1*. The prediction will be the entity that needs to be extracted from the document to answer the question.

## Information Retrieval

Around 3000 BC, the Sumerians created specific locations to preserve clay tablets bearing cuneiform inscriptions, which is when the practice of archiving written knowledge began. The Sumerians understood even then how important it was to meticulously organize knowledge and access the archives to use it efficiently. To categorize each tablet and its contents, they created unique categories.

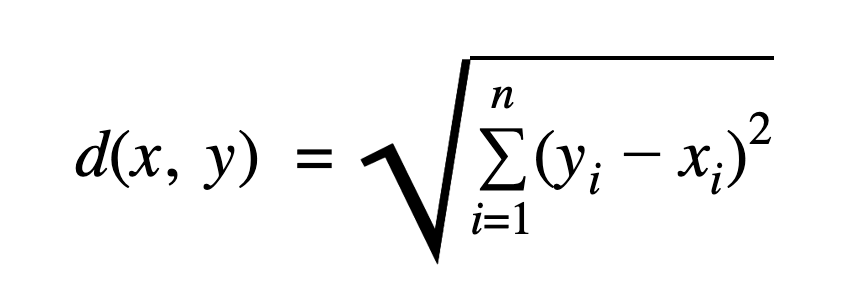
With the development of the printing press and paper throughout the ages, the demand to preserve and access written knowledge grew. The ability to store and automatically retrieve vast quantities of data was discovered shortly after the invention of computers. The concept of automated access to vast volumes of stored knowledge was first introduced by Vannevar Bush in his groundbreaking 1945 article "As We May Think" [22]. This concept came to life in the 1950s with more specific explanations of how text archives may be automatically searched. The fundamental concept of using a computer to search text was expanded on in several publications in the middle of the 1950s. In 1957, H.P. Luhn presented one of the more impactful approaches, suggesting that words be used as document indexing units and that word overlap be measured as a retrieval criterion [23]

The field of Information Retrieval (IR) was born in the 1950s out of this necessity. Over the last forty years, the field has matured considerably. Several IR systems are used daily by a wide variety of users. This article is a brief overview of the key advances in the field of Information Retrieval, and a description of where the state-of-the-art is at in the field [22].

Information retrieval module is used to find the article whose titles and contents relates the most to the query. This is done by using several preprocessing techniques on the text and on the query that will help identify the document. The following techniques have been applied on each text and on the query:

* Abbreviation replacement: Abbreviations like U.S.A. transformed into the full text United States of America.
* Number elimination: Since we are not interested in dates when it comes to information retrieval, numbers can be eliminated from the documents and from the query.
* Separators removal: Text separators are essential for enhancing textual content's readability, comprehension, and organization in both printed texts and digital interfaces. They act as a visual aid for readers, assisting them in finding pertinent information fast or recognizing different portions within a longer text. No information gain comes from text separators, so they were removed.
* Removal of stop words: Stop words are the words in a stop list (or stoplist or negative dictionary) which are filtered out (i.e. stopped) before or after processing of natural language data (text) because they are deemed insignificant [24]. There is no single universal list of stop words used by all natural language processing tools, nor any agreed upon rules for identifying stop words, and indeed not all tools even use such a list. Therefore, any group of words can be chosen as the stop words for a given purpose. The "general trend in [information retrieval] systems over time have been from standard use of quite large stop lists (200–300 terms) to very small stop lists (7–12 terms) to no stop list whatsoever" [25].

After the preprocessing part is done, a frequency vector will be computed for each document and the query, followed by the Euclidian *(Figure3.4)* distance between each article and the question itself, where “y” is the frequency of one word in the article and “x” is the frequency of the same word in the query.



*Figure 3.4: Euclidian distance formula*

## Answer Extraction

The last module of the question answering system is the answer extraction part. This module has the role of identifying the answer of the input query passed by the user and to return it.

### Keywords extraction

A task in Natural Language Processing called "keyword or key phrase extraction" involves examining a text or proposed document and using various algorithms to return a list of words or phrases that are pertinent to the input document and can be used as the document's keywords or key phrases. The keywords that are extracted from a text can also be used to make the categorization process for search engines easier. Rather than searching through every document for every word that is needed, the engine can scan each document's keywords and choose those that partially match the desired words, which speeds up the search process.

Keyword extraction is a very important task when it comes to answer extraction. They will greatly help us when it comes the sentence scoring and finding the most relevant sentences that will have the answer to our question.

### Preprocessing

The document and keywords will go through a preprocessing step before the answer can be extracted. This step is crucial to extract an accurate answer. First thing first, all separators will be eliminated from every sentence since they are not needed when it comes to gaining information about what can be found in a particular sentence. After separators are removed, the following step is the removal of stop words since they give next to no informational gain. By removing them our system will have to go through fewer words meaning faster and more accurate results. The next step is to find the lemma and stem of each word; this step will assure that all the words will be reduced to their root form based on the context and no mismatches can occur. For example, “assassinated,” will be transformed into “assassin” which will ensure that for the question “Who assassinated Lincoln?” we will find the sentence that talks about his assassin if it exists in the document.

### Relevant sentences and answer extraction

After all the preprocessing is done and we retained as much information as possible in little space as possible, the step where sentences are scored is next logical step. This will help the system narrow the sentences that are relevant to the question, and it will make our tool be more likely to find the answer quicker and more accurately. The process is composed of two simple metric: Same word sequence score and Matched keywords score [20].

Same word sequence score is a metric that will iterate through every word of a sentence and compare it to the keywords, if the word can be found in the keywords, then we will go to the next word until we do not find a keyword anymore. Every time we find sequential keywords we will take note of that and the largest sequence present in the sentence will be stored as a sum of keywords found. Each keyword can be found once per sequence.

Matching keywords score is another metric used to help us determine which sentences are more important for the query. Every time a keyword is found in the sentence, we will take note of that, and a counter will be raised. The maximum score for a sentence will be the number of keywords, meaning that a keyword will be counted only once even though it can be found multiple times in the sentence.

After both metrics have been computed, they will be added for each sentence and only the best scores will be considered as possible answer candidates to the question entered by the user. Those candidate answers will go through named entity recognition and will try to find a match between the result of the SVM prediction of the question classification and the result of the named entity recognition. If a match has been found (ex SVM predicted “PERSON” as the answer needed and a “PERSON” was found in the sentence), then we will use dependency parsing on that entity to find which word is related to. If any of the related words is a keyword, then we have an answer and we will return it. In case no matches have been found in any of the sentences, then the most likely answer is the first sentence with the highest score and we will return it.

# Software Implementation

## Technologies used

### Python

Python is a dynamic high-level programming language created by Guido van Rossum from Netherlands. This programming language was released in 1991 and was designed with the idea of being easy to use. Its name comes an english comedy group “Monty Python.” Python programming language uses a simple syntax, this can be further supported by a phrase “pythonic” that is used in the python community. This phrase refers to code that is written based on the minimalistic and more visible philosophy which can be easily done using python. On the contrary, code written in this language the is hard to understand or looks similar to other programming languages is considered to be “unpythonic” [26].

A quick summary of Pythons philosophy can be found in “The Zen of Python” [27]. Some of the python principles included there are:

* Beautiful is better than ugly.
* Explicit is better than implicit.
* Simple is better than complex.
* Complex is better than complicated.
* Readability counts.

Python includes the possibility to write procedural programming, imperative programming, functional programming and it can also be used in object-oriented programming. Unlike other languages such as Java, C++, Python variables do not need to be declared. Their type will be determined by the interpreter based on what they store or the type of operations performed with them. By being a dynamic programming language, Python can manage the garbage collection part himself and it also has a high level of abstractization.

### Pycharm

Pycharm[28] is an integrated development environment released in Febrary 3rd 2010 by the Czech company JetBrains for the programming language Python. This environment has a lot of tools integrated to make it easy to use for developers. It also supports web development by using the framework Django.

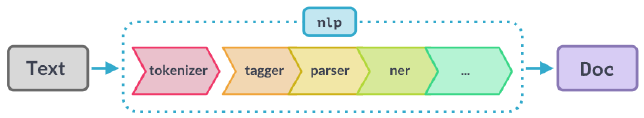
By being a cross-platform application, PyCharm can be used with Windows, macOS, Linux and comes with 3 versions:

* Community: A free open-source version which offers tools for writing code, refactoring, visual representation, and application versioning.
* Professional: A payed version used for development of professional software. Includes everything mentioned in the community version and more. New things that can be accessed by paying are support for popular web development frameworks (Django, Flask), databases support, and tools for Big Data.
* Edu: A free open-source version used for teaching students.

### SpaCy

SpaCy[29] is an open-source library used for natural language processing in Python. This library has been designed to be used in production and with its help, computers can process and “understand” a huge quantity of text. SpaCy can also be used in information retrieval systems and also in machine learning.

While some components of spaCy can work independently, others need trained pipelines to be loaded to allow spaCy to predict linguistic annotations such as part of speech recognition (verb, noun, etc.). This pipeline is composed of a tokenizer (all pipelines need to include a tokenizer) whose main purpose is to segment text received as input into tokens. This can be followed by other components that vary depending on the type of pipeline used and the developer's preferences. The programmer has the options to enable/disable components.

*Figure 4.1: spaCy pipeline*

SpaCy offers 4 packages for English (en\_core\_web\_sm, en\_core\_weh\_md, en\_core\_web\_lg and en\_core\_wed\_trf). Even though the difference in sizes is significant (12 MB for the smallest one to 438 MB for the biggest one), the performance seems to be similar for all the packages (oly about 1-2%). For this reason, we chose to use the smallest package available en\_core\_web\_sm. This will lead to a faster computational speed of the system.

The most important components used from this package are:

* Tagger: Which predicts the part of speech of a token.
* Parser: Which will predict the semantic dependencies between tokens.
* Lemma: Transforms the token into its dictionary form.

### NLTK and WordNet

Natural language toolkit or NLTK [30,31] is an open-source library used for natural language processing in Python. From this library the most important modules used are WordNet and Stemming.

WordNet is a lexical database which contains semantic relationships between words for more than 200 languages. Since the main relationship between words present in WordNet is that of synonyms (words that describe the same thing and can be interchangeable), they will be grouped in unordered sets called synsets. WordNet database consists of approximately 155.000 words that are organized into 175.000 synsets and all of the synsets are tied together using semantic relationships.

The most frequent relationship between those synsets is the hypernym relationship, which ties together words like “color” to more specific words for example “red.” Using this relationship WordNet can specify that the word “red” is included in the “color” category. This creates a tree-like structure between the relationship of the hypernyms in which every parent of a node represents the direct hypernym. If we iterate through the tree of any noun from the database, we will reach the root (“entity”).

### Sklearn

Sklearn [32] (scikit-learn) is a free open-source machine learning library created to be used for Python. It was designed by the French data scientist David Cournapeau and it was released in June 2007. It is a cross-platform library meaning that it can be used with different operating systems such as Windows, Linux and macOS. In November of 2012, scikit-learn was descrbied as one of the most popular and well-maintained libraries. Also, in 2019 scikit-learn was noted as the most popular and used machine learning library in GitHub. This library features various classification, clustering, and regression algorithms which include k-means, random forests, gradient boosting, and support vector machines.

In our system, sklearn was used to create, train, and test a support vector machine that will classify a query. The result of the query is the type of answer that our system needs to return to the user.

## Workflow of the system

The objective of the thesis is to create a question answering system that will return a relevant answer to an input (query) from the user. To achieve this, four crucial steps are required: SVM training, Question classification, Information retrieval and Answer extraction.

The first step is to train and test an SVM so that it will be able to predict the answer that needs to be retrieved. To do this, we will iterate through each question that training dataset and we will extract the following features from them:

* Unigram
* Bigram
* Headword
* Hypernym
* Query expansion

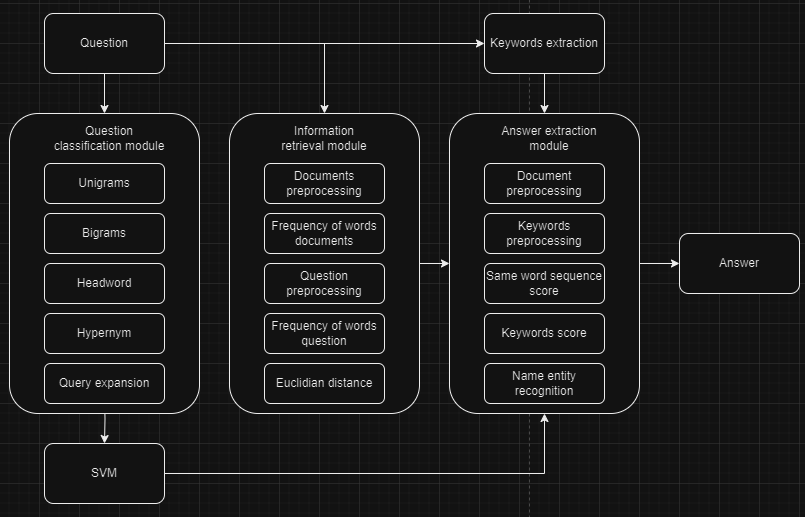
All the features extracted from the available question will be stored and then fed to the SVM. A split of 80-20 has been used to train the SVM, along with the option to shuffle the data every time it is trained and to have an even distribution of data. These options ensure that the SVM will have the best chances to be properly trained.

The second step of our system is to classify the input query from the user. Once the SVM has been trained, the question of a user will go through the same process as the training process of the SVM. This is done to extract the same features on which our support vector machine has been trained on, ensuring that the prediction will be as accurate as possible.

The next step following the prediction for the answer is to find the document that is most likely to have our answer. This step uses a lot of information retrieval techniques such as number elimination, separators elimination, abbreviation replacement, stop words removal, etc. This preprocessing of the text and query is necessary in order to ensure that only the words that give the most information gain remain. After preprocessing, we should have only relevant words and the calculation of Euclidian distance can begin. To compute the distance between a document and the query, we must create a frequency vector for all the articles in the dataset and the query. After the matrixes of frequencies have been computed, we will find the smallest distance between a document and the question, the result will be stored for further usage.

The last step of the system is to extract the answer from the document categorized as relevant. Before doing so, we need to extract the keywords of the query because we will need them later. When the keywords are extracted, we will pass them alongside the document to the answer extraction module. The features will be passed to the module and will go through a preprocessing step to ensure that only relevant information will be extracted. This preprocessing step includes ideas from the previous module (Information retrieval) which are: separators removal and stop words removal; but also have new things added such as: lemmatization and stemming of each word. After the preprocessing step is done, a score for each sentence is computed by using the following metrics: same word sequence score (the biggest sequence of unique keywords) and key words score (sum of unique keywords found in the sentence). This score will help us determine the candidate sentences that can contain the answer for the question. Lastly, we will use only the highest scored sentences that will go through named entity recognition and if we find an entity that matches the SVM prediction we will check if that token has a relation with any of the keywords extracted. If the token is related to any of the keywords, then it will be returned as the answer. In the possible case where we do not find an entity that matches the SVM prediction, then we will return the first sentence from the highest scored sentences pool, because it is most likely to contain the answer for the question.

## Application diagram

*Figure 4.1: Question answering system diagram*

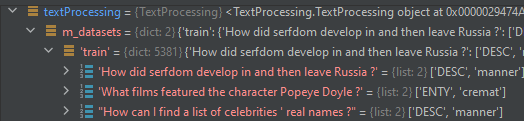
## SVM training

Before making the actual system, an SVM needs to be trained to predict the input of the user. In our case, we chose a linear SVC for the task. At the start of the application, an object “textProcessing” of classs “TextPreprocessing” will be initialized and will be passed to the function called “getTrainTestFeatures.”



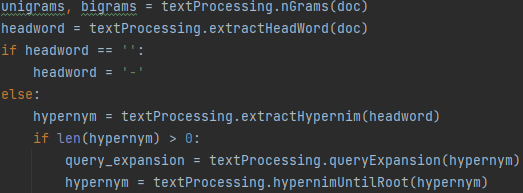
*Code 4.1: Calling the main function for training and testing*

In this function there are two lists of features and two lists of ground truth: training\_feature\_list, testing\_feature\_list, true\_positive\_training and true\_positive\_testing. After which the SVM will be initialized. The next step is to read the TREC dataset and to store it. We chose to store the dataset in a dictionary, where the key will be our question and the value will be the type of answer it requires. The reading process is done by calling a method from “TextProcessing” class, called “readData().” This method will store it as an attribute of the object “textProcessing.”



*Figure 4.2: Dictionary of TREC dataset*

After the TREC dataset has been read and stored, we will go through each question and start to extract the features needed to train the model. We create the appropriate variables for each feature that will be extracted: lists (“unigrams,” “bigrams,” and “query\_expansion”), string (“headword” and “hypernym”); and we call the methods from TextProcessing class that will populate the variable created.



*Code 4.2: Extracting the features for SVM*

### Unigrams and bigrams

Unigrams are created by iterating through the text and appending every word into a list that will be returned. Bigrams are extracted in a similar way to unigrams, but this time we will append the current word and the next word separated by a “-” into a list.

*[‘How’, ‘did’, ‘serfdom’, ’develop’, ’in’, ’and’, ’then’, ’leave’, ’Russia’, ’?’]*

*[‘How-do’, ’do-serfdom’, ’serfdom-develop’, ’develop-in’, ’in-and’, ’and-then’, ...]*  
*Example 4.1: Example of how unigrams and bigrams look*

### Headword extraction

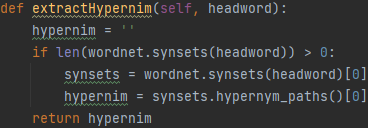
The headword of a sentence is the word that contains the highest amount of information, and it will have a crucial role in the training and testing of the SVM. Headwords will be determined by the syntactic parsing of a question and applying a set of rules. The rules used to determine the headword of a sentence are the following:

* If a subject is present in the question we will check its part of speech:
  + If it is a common/proper known or an adverb, it will be chosen as the headword.
  + If it is a personal pronoun, we will search for the direct complement of the question and its part of speech:
    - If it is a common noun or an adverb, we will choose it as the headword.
    - If it is a proper noun, we will use “name” as the headword.
* If there is no subject in the question, we will search for its attribute and if the attribute’s part of speech is a common/proper known or an adverd, it will be chosen as the headword.
* If the question has no attribute, then we will choose the direct complement as headword.
* If none of the above conditions are met, then we choose the first noun as the headword.
* Lastly if no headword can be extracted, then we signal it in the “headword” variable and code that is tied to the headword will not be functional anymore.

### Hypernym extraction

Hypernyms are words that are more general, and they include the meaning of other words that are more specific, for example the hypernym of the word “red” is the word “color.” Since some of our training questions can be too specific, we decided to use the hypernyms of the headword. This approach will lead to a more generalized question and will help the training process of the SVM. To make the hypernym extraction process easier, we decided to use NLTK library, more specifically the WordNet module.

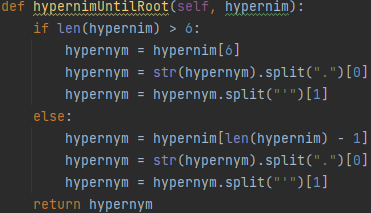
Extraction of hypernyms is done in “TextProcessing” class by the method called “extractHypernym.” Here, using the method from WordNet called “synsets” we will extract a list of hypernyms for the headword. From the resulted list, we will consider the first element to be the most relevant meaning of the word. As an example, we get the word “serfdom” as the headword, in this example the most relevant hypernym will be “bondage.”



*Code 4.3: Hypernym extraction*

In WordNet, the words are stored in a treelike structure, meaning that every parent node of a word is its hypernym. In *Code 4.2* we can observe the usage of a method called “hypernym\_paths().” Using this method, we generate a treelike structure that will lead to the root of the semantic tree. As an example, the root of every noun will always be “entity.”

We could use the hypernym of the headword and continue from there, but since not all head words are specific enough, choosing the direct hypernym can minimize the chances of a match between two sentences whose headwords can be found on the same path of the hypernym tree. This chance is small because the two words can be found on different levels within the tree. Because of this, we decided to implement a small logic that will help with the problem. Every time we will check the length of the hypernym tree returned by the function “hypernym\_paths”. If the list's length is bigger than 6, the word is not general enough, so in this case we will choose the 6th hypernym in the list that will replace the headword. In contrary, if the length is smaller than 6, we consider the word to be general enough and, in that case, we choose his direct hypernym. This step is done by calling the method “hypernymUntilRoot.”



*Code 4.4: hypernymUntilRoot function*

### Query expansion

This feature is closely related to the headword. Meaning that if no headword was found, then this step will be skipped. It is also similar to hypernym extraction, but this time we get all the hypernyms until root and will give them a weight. If a headword is found, then we will use again the method “hypernym\_paths” on the headword.

We will save the list and we will process it a bit. Since the first element of the list is the root of the headword and the last element is the direct hypernym, we first reverse the list. In *Example 4.2* we can see how an element of the hypernym list looks like.

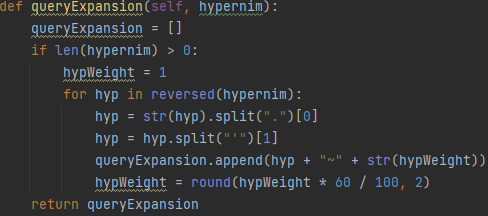
*["Synset('entity", 'n', "01')"]*

*Example 4.2: How a hypernym looks like*

Having the list of hypernyms ordered from the direct hypernym of the headword to the root, we can start processing it until we remain with only a list of strings. As we can see in *Example 4.2* the word that we need is located after the character “ ’ ”. For us to access this word, we need to split the method using the character we need to split it by. This will result in two strings and the second string will be the word needed. After saving the word we will concatenate it with the character “~” and its weight. The weight represents the distance of the word to the headword, meaning that direct hypernym will have a weight of 1. We decided to lower the weight by 60% for every iteration of the hypernym tree. This is done because the further we are from the headword, the general the hypernym becomes, meaning the less information we can retrieve from it.

*['serfdom~1', 'bondage~0.6', 'subjugation~0.36', 'relationship~0.22', 'state~0.13', 'attribute~0.08', 'abstraction~0.05', 'entity~0.03']*

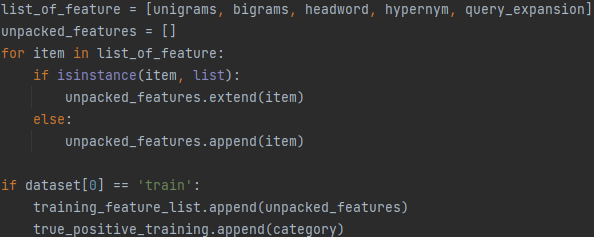
*Exampled 4.3: Query expansion*



*Code 4.5: queryExpansion method*

### List of features

After all the features have been extracted, we will store them into a list of strings, alongside a category list, those lists will be fed to the SVM model to train on them. Because not all our features have the type needed (some of them are lists of string), we need to change them to fir our needs. To do this we will save all the features in a list called “list\_of\_features,” and we will iterate through it. If the element is already a string, we simply save it into a list called “unpacked\_list” but, whenever we find an element that is a list we will break it into separate strings and then save it. When all the features of a sentece are saved along with the category, they will be stored in two separate lists “training\_feature\_list” and “true\_positive\_training.” Those lists will be fed to the training function of the SVM.

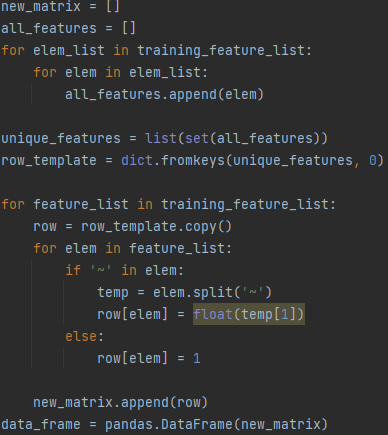
*Code 4.6: Creation of feature list and category list*

### SVM training

SVM training is done inside the function called “trainAlgorithm” which takes the following arguments: training\_feature\_list, true\_positive\_list, and the SVM itself.

Since the SVM needs a matrix of features, we need to adjust our data to successfully send it to the model. To do this the first step is to iterate the training\_feature\_list and save all the elements into a new temporary list called “all\_features.” After that, we will convert the temporary list into a set. Because this structure needs to contain only unique entries, all the duplicates will be eliminated. When we have only unique entries, we will convert it back into a list since we need this type of structure in the next step.

Next step is to create a dictionary “row\_template” with the keys being every element of the list and the value of each key will be 0. When the dictionary is created, we can iterate through the training\_feature\_list once more. On each iteration we will create a copy of the “row\_template” and this copy will be store in a variable called “row.” We go through every element of a sentence and if the elemnt contains the character “~,” then we will use split method to get the weight that was attributed since this is a hypernym and will change the value of the dictionary where the element can be found from 0 to the weight. In case the special character is not present, then we simply change the value from 0 to 1. After a complete iteration of a sentence the resulted dictionary will be stored in a list of dictionaries called “new\_matrix.” This process happens until there are no more elements in training\_feature\_list to go through.



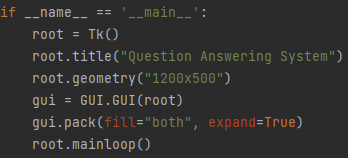
*Code 4.7: Preparation of the feature matrix needed for SVM*

Lastly, after the matrix of features is done, we can start the training of the SVM. We will use some special parameters to help the model train better. The first parameter is “test\_size” which specifies how the data will be splitted. We chose a standard split of 80-20, meaning that that 80% of the data will be used to train the algorithm and 20% of the data will be used for testing. Next parameter is shuffle, used for cross validation. This parameter ensures that the data will not be the same for every training iteration, meaning that it will prevent unintentional ordering biases. Shuffle is important when it comes to training a machine learning algorithm since it can prevent the model from learning based on the ordering of the data. The last parameter is stratify, this one is also important, because it will ensure that all the possible classes are distributed proportionally both in training and testing.

After the SVM has been successfully trained, the model will be saved in a pickle file for later usage when it comes to question classification.

## Graphical interface

Graphical interface or GUI was created using tkinter toolkit from Python. In the main function we created an object called “root” of type “Tk”. We populated the object with the title of the application (“Question Answering System”) and with the dimensions of the system (“1200x500”). The behavior of the graphical interface will be handled by the class called “GUI.” To use this class, an object of that type called “gui” is defined. Lastly “mainloop” function was called using the object “root,” this will keep the application “alive” until the user closes it.

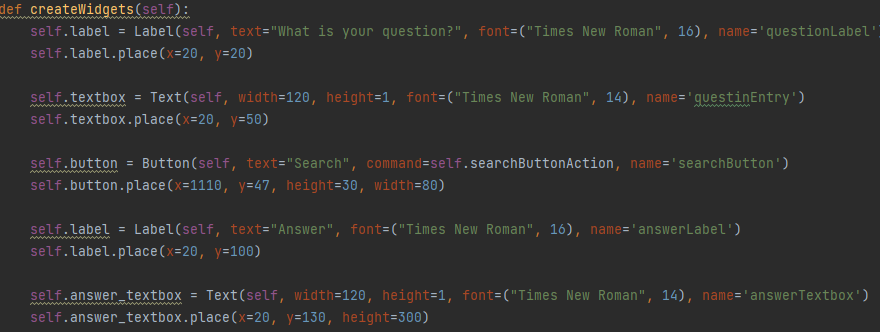


*Code 4.8: Creating the GUI*

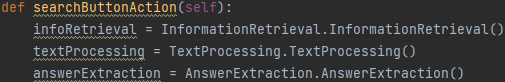
### GUI class

This class is composed of two main methods: createWidgets and searchButtonAction. They will handle the creation of widgets and how they will behave.

CreateWidgets is a simple method where all the widgets will be initialized. First, we will initialize a label called “questionLabel,” and will indicate where the user needs to input the question that needs to be answered. Below this label a textbox is created, here the user of the system will write the question in natural language. From this textbox the question will be extracted and used to find the relevant document and answer. On the right side of the textbox a button called “searchButton” will be initialized. By pressing this button, the question answering will begin to serach the document, find the answer, and extract it. Once the answer was extracted, it will be shown in a textbox called “answerTextbox”.

*Code 4.9: createWidgets method*

SearchButtonAction method, handles how the button called “searchButton” behaves. When it is clicked the system will go through all the steps needed to find the answer. To do this, the first step is to created object of the following classes: InformationRetrieval, TextProcessing, AnswerExtraction.



Code 4.10: Creation of the needed objects

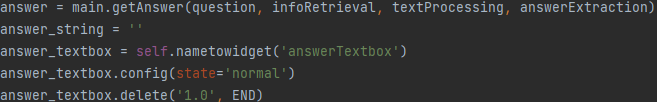
After this, the text that was entered in the “questionEntry” textbox will be saved into a variable called “question.” To extract the question from the textbox, we need to declare a variable, in our case “textbox” and we will link it to “questionEntry” using the function called “nametowidget.” This will ensure that “textbox” variable has access to the attributes of “questionEntry.” To extract the text entered by the user, we will use the “get” function with the following parameters: “1.0” and “end-1c.”

The first parameter (“1.0”) represents the position of the element that needs to be retrieved, in our case 1 refers to the line and 0 refers to the column. The second parameter (“end-1c”) specifies what to retrieve from the specified position. In our case “end” refers to the end of the textbox and “1c” will exclude the last character. We need to exclude the last character because the new line character (“\n”) is always added at the end of a textbox.

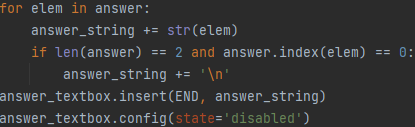


Code 4.11: Extraction of the question from textbox

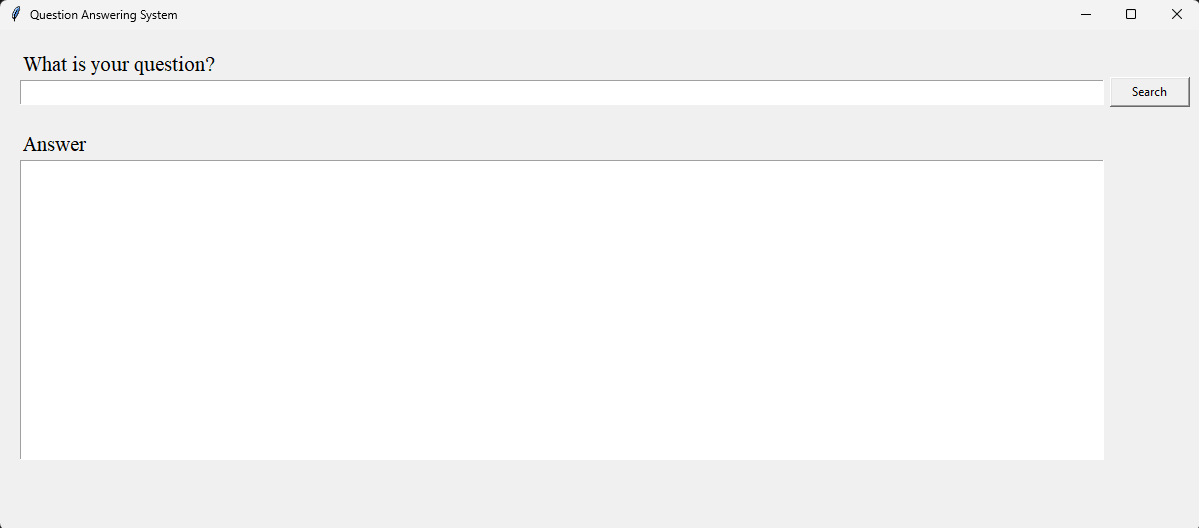
Once the question has been extracted, we will call the function “getAnswer” and the resul will be stored in a variable called “answer.” We designed the system to retrieve the answer and the sentence where it can be found if the predicted entity is found, otherwise it will return the sentence where the answer might be found. Because of this, the variable “answer” will be a list of strings which cannot be passed to a textbox, so we create a variable “answer\_string” where we will concatenate the result. Before the concatenation, we need to link the the “answerTextbox.” This process is done once again using the “nametowidget” function. After the linkage is done, we will enable the possibility of writing in the textbox, and we clear it in case text is present from previous usage.

*Code 4.12: Answer extraction and linkage to answer textbox*

Now the process of concatenation can begin. We will iterate through every element that cand be found in the “answer” variable. First thing is to give the store the first element found in the “answer\_string” variable. The element will be passed through a cast function to ensure that the type is correct (string). Then we will check if the answer length is greater than 2. If it is, it means that an entity has been found and in the “answer” variable we have the answer along with the sentence where the answer has been extracted from, so we need to add the character “\n” at the end of the first element to separate the answer from the sentence. At the end we will insert the “answer\_string” variable in the textbox and we will mark it as read-only. So, the text box cannot be modified unintentionally.



*Code 4.13: Sending the answer to the textbox*

*Figure 4.3: GUI of the system*

## Question classification

After the SVM has been trained, the question classification module can begin. After the “Search” button has been pressed, the “getAnswer” method is called. The first thing done in this function is to use the SVM to predict what type of answer is needed for the entered question. This is done by calling the “predictQuestionCategory” function, which takes the question as the input. The result of the function will be retrieved in a variable called “prediction\_result.”

In “predictQuestionCategory” function, we will load the SVM model that has been trained. After loading the model, the question will go through the same feature extraction process mentioned in section **4.4 SVM training.** As a short recapitulation, we will extract the unigrams, bigrams, headword, hypernym, and query expansion features from the question. These features will be stored into a list of features and later we will create a matrix of features. This matrix of features will be fed to the trained model to make a prediction. The prediction will be returned and stored in the variable “prediction\_result” mentioned in the previous paragraph.

An example of a result that will be retrieved from the SVM is [‘ind’]. This type of result will be predicted from questions like “Who assassinated Lincoln?.” Ind refers to individual, meaning that for the question in the exampled, the expected answer should be an individual.

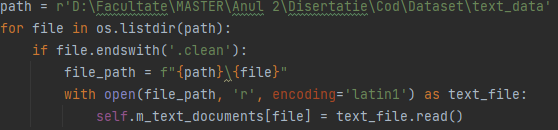
## Information Retrieval

After the answer needed is extracted, we will find the most relevant document based on the question of the user. The object “infoRetrieval” will be used to call the appropriate methods needed to get the document.

### Document and question processing

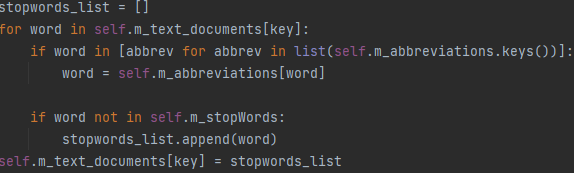
This process is done to eliminate all the unnecessary elements present in the documents and the question. It is a crucial step that will ensure the best possible result.

First all the documents will be read and stored in a dictionary of documents called “m\_text\_documents.” This dictionary will have as a key the document's name and the value will be the text itself. To do this, the path of where the articles can be found is needed. We go to the path and will check every document that is found there. If the document ends with “.clean” then we will open it and read its contents.



*Code 4.14: Reading and storing the documents*

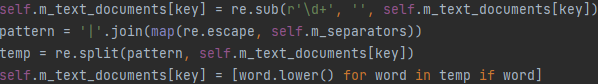
When all the articles have been stored, we will go through each one and will start the cleaning process. The first step is to replace abbreviations of words to their full form. Words like “U.S.A” will be replaced with “United Stated of America”. To do this, we will take each word from a document, and we will check if that word is present in a dictionary of abbreviations. If the result is positive, we will change to word with the value of the dictionary whose key is our abbreviation. In the same iteration of a document, we will also remove the stop words. A stop word is considered to have next to no informational gain, because of this there is no real reason to keep them. We will eliminate these stop words using regex.



*Code 4.15: Abbreviation and stop words removal*

The next step is to eliminate numbers and special characters from the document. Numbers were eliminated because there are other elements in a text/question that have more information than numbers. If we take the following text as an example “I On 31 January 1949, during the Chinese Civil War, Communist forces entered Beiping without a fight.” Here we see numbers that represent dates. These dates will never give the system more information about what is the text about compared to words like “Chinese,” “Civil,” “War,” etc. For this reason, numbers were eliminated. Same can be applied for the special characters like separators. Our system does not need them to get the information about what a text is about, so we remove them.

To achieve number and special characters removal, we used regex library once again. The results will be stored again in the dictionary of the documents.

*Code 4.16: Removal of numbers and special characters*

Once all the sentences of an article have been processed, a list of unique words “m\_unique\_words” will be created based on the results. The list will be updated for every document in our dataset and only documents, the question will not affect this list. After all documents have passed the processing step, and the list of unique words is finalized, the elements will be sorted in an ascending order.

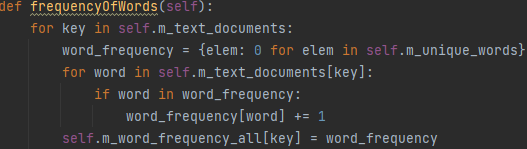
*{‘S08\_set1\_a1.txt.clean’: [‘kangaroo’, ‘marsupial’, ‘family’]}*

*Example 4.3: Dictionary of documents after processing*

The question will be passed through the same process as all the documents from the dataset. As mentioned in the paragraph before, the result from the processing of the question will not affect the list of unique words.

### Vector of frequencies

After the question and documents have been processed. The next step is to compute the matrix of frequencies. This matrix of frequencies will help us determine what document is relevant to the question. To compute this, we will again go through every document and create a temporary dictionary “word\_frequency” that has the same dimensions as the list of unique words “m\_unique\_words.” Word frequency dictionary will have each unique word as keys, and we will give each key the value 0, this value will represent the number of times that word can be found in the document. After the temporary dictionary has been initialized, we will iterate through every word in the article. If the word can be found in the list of unique words, then the value of the dictionary whose key is our word will be increased by one. After all the words in a document have been parsed, then we will store the matrix of frequencies in a new dictionary called “m\_word\_frequency\_all” whose key will be name of the document and value will be the matrix itself.



*Code 4.17: Frequency vector creation*

*{‘S08\_set1\_a1.txt.clean’: {‘abandon’: 0, ‘abandoned’: 0}}*

*Example 4.4: Matrix of frequency*

### Euclidian distance

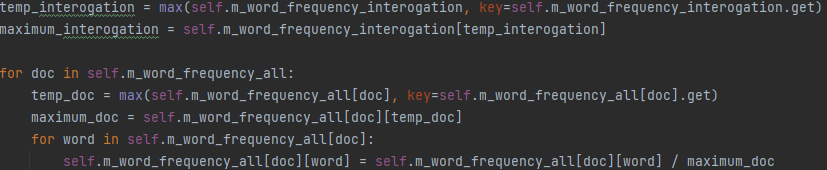
Euclidian distance is the algorithm that will help the system return the most relevant document based on the question input of the user. In mathematics, the Euclidian distance between two points is represented as the segment that separates those two points in the space. In the Euclidian plane, having two points X with the coordinates () and Y with the coordinated () then the distance between those two points can be determined by the following formula:

*Formula 4.1: Euclidian distance*

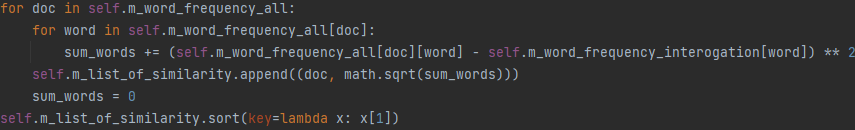
Following this formula, in our case the Euclidian distance for one document and the question will look like this:

*Formula 4.2: Euclidian distance for one document*

After using this algorithm, the document with the score closer to 0 is the closest to the question. This means that it is the most relevant document that will most likely contain information about the query. Before computing this distance, we need to normalize the frequency vector. To do this, a simple division will be made. We will go through each dictionary of frequencies and find the highest frequency score. Once found, we will divide all the frequencies to the highest one and update the values.

*Code 4.18: Frequency vector normalization*

Once everything has been normalized, we can apply the Euclidean distance formula (*Formula 4.2)* for our frequency vectors.

*Code 4.19: Euclidian distance applied*

The results will be stored in a variable called “m\_list\_of\_similarity” and sorted. After sorting, the first element of the list will be the most relevant document found.

## Answer Extraction

### Keyword extraction

Before the answer extraction module can begin, we will need to use one method from TextProcessing class, this method is “extractKeywords.” This method will take one argument, this argument being the question.

The first keywords that will be extracted are the ones in quotation marks if they exist. Since in our dataset we can find characters such as “ '' “ and “ `` “ used as quotation marks, before extracting the keywords we need to find this characters and replace them with “ “ “. This replacement will be done by using the method “replace” as we can observe in the code bellow.



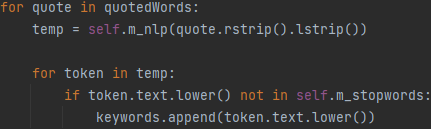
*Code 4.20: Replacement of characters*

To identify words that are between quotation marks, we will use the method “re.findall” from regex with the following argument “([^”]\*)”. The string used as parameter is an expression and is used to recognize a particular pattern in a string. This pattern will identify any string which contains the character “ '' ”. This method will return a list of string or strings that can be found between this type of characters.



*Code 4.21: Finding expressions between quotation marks*

Once we extracted the words, we will iterate through “quotedWords” list and we will pass it through spaCy pipeline to segment each token individually and save them in the variable “temp”.

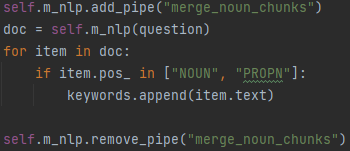


*Code 4.22: Segmentation of words in quotation marks*

We will go through every token, and for every token found we will check if the lowercase of it can be found in our stop words list. If the token is not a stop word, then we will save it as a keyword.

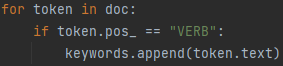
Next step is to get all the common and proper nouns that can be found in the question, along with the words that are semantically dependent of them. To make this happen, we will add “merge\_noun\_chunks” to the spaCy pipeline. This will group all the tokens whose part of speech is a noun with the tokens that are dependent of them. As an example, if we use the question “What is the oldest profession” in the classic spaCy pipeline the result will be [What, is, the, oldest, profession, ?], but if we use the “merge\_noun\_chunks” component the result will be [What, is, the oldest profession, ?].

After we add the new component into spaCy pipeline, we will segment the current question and we will save the returned tokens in a variable called “doc”. Then, we will go through this list of tokens and check if the part of speech is either a common noun or a proper one using the “pos\_” attribute. If the condition is true, then the token will be saved into “keywords” list.



*Code 4.23: Common/proper nouns keywords*

The last type of keywords that are extracted are verbs. They will be extracted by using the attribute “pos\_” again. We will check every token from the segmented question. If the part of detected is “VERB” then the word will be added into the keywords list. At the end of the function, we will return this keywords list since we will need it to find the answer of the query.



*Code 4.24: Verbs extraction*

### Relevant sentences

After keywords have been extracted, it is time to call the function that will retrieve the most relevant sentences. To do this, we call the function “findRelevantSentences” from the class called “AnswerExtraction” with keywords and the article as the parameters.



*Code 4.25: Calling findRelevantSentences*

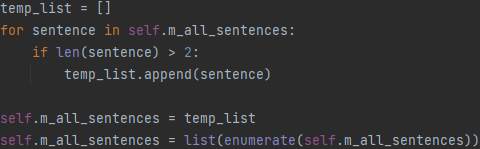
Once inside the function, we will declare our separators list (separators were defined by us) and a list of stop words (list used from NLTK). The next step is to read the most relevant document and store its content into a list “m\_all\_sentences”. This list of lists will contain the following:

* First element is the index of the sentence (used later to return the sentence where the answer can be found).
* Second element will be the actual sentence.

Before we calculate which, sentences are relevant to the question, the sentences need to go through a processing step.

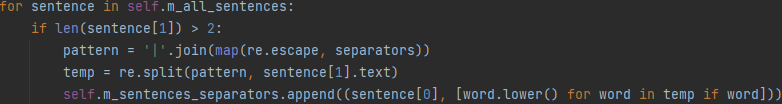
#### Sentence processing

First step is to iterate through all the sentences and check if the sentence length is greater than 2. Because some documents contain special characters, for example Chinese ones, the reading process can produce weird sentences that contain only one or two words. We decided that if that case happens, those sentences will not have the answer and because of this, they are eliminated. Every sentence that has a length greater than two will be saved in a temporary list. After all the sentences passed through this filter, we will update the “m\_all\_sentences” with the temporary list created.



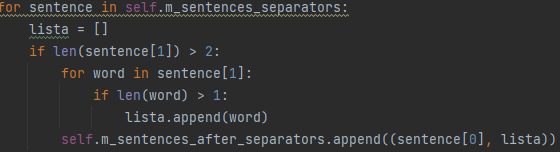
*Code 4.26: Sentences smaller than two removed*

The next step is to get rid of the separators defined in the variable “separators”. To do this we will use regex. To do this we will iterate the sentences, we check its length and if the length is greater than two we will create a regex pattern and store it. This pattern is created using the function “.join”. Every separator defined by us will be concatenated and separated with the or symbol “|”. Once the pattern is created, we will use it in a sentence. If a separator is found, then we split it and save only the second part of the split (this will contain the sentence) and store it into a temporary variable. After the split we will save the result in a new list called “m\_sentences\_separators.”



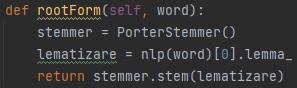
*Code 4.27: Separators elimination*

After the sentences list has been updated, we will eliminate the stop words from it. We do this by checking all the words in a sentence. If that word is not present in the NLTK stop words list, then we will save it in a temporary list. Once all the stop words are eliminated from a sentence, we will tokenize every word and save the result in a new list “m\_sentences\_after\_separators.”



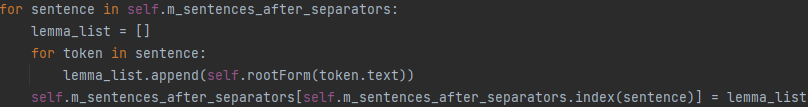
*Code 4.28: Elimination of stop words*

Next, we will use a combination of stemming and lemmatization to change every word to its root form based on the context of the sentence. To do this we created a function called “rootForm” which takes a word as a parameter. This function uses “.lemma\_” attribute from spaCy and “.stem” function from NLTK.



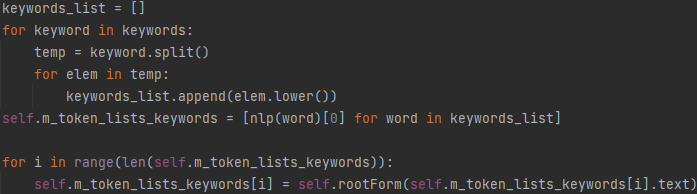
*Code 4.29: rootForm function*

Going through all the sentences and calling rootForm function for every word of the sentence will retrieve us sentences with all the words transformed into their root forms. After a sentence has been transformed, we will update the list “m\_sentences\_after\_separators.”



*Code 4.30: Root form of sentences*

Once this process is finished, we will take all the keywords, split them in case there are merged chunks from the extraction, change them to lowercase and tokenize for the stemming and lemmatization process. After updating the keywords to their root form, we are ready to start scoring all the sentences



*Code 4.31: Keywords processing*

#### Sentence scoring

To find the most relevant sentences, a scoring method was used. This method is composed of finding the longest sequence of unique keywords in a sentence and computing the number of unique keywords found in a sentence.

To compute the longest sequence of unique keywords we will iterate through every sentence. A counter called “matching\_sequence\_count” will be initialized for every new sentence. Next, we will take every word of said sentence, and check if it can be found in the keywords list. If the word is a keyword, a temporary counter “temp\_counter” will be initialized with the value 0 and a copy of the keywords list called “copy\_token\_list\_keywords” will be made. Now we will start to count from the word found in keywords onwards. If we find a keyword, the counter will be increased by 1, and we will remove the keyword from the copy list. If we do not find a keyword, we will remember where the last word checked is in the sentence and we will update the “matching\_sequence\_count” with the biggest sequence of keywords found. For example, let use the question “Who assassinated Lincoln?”, the keywords for this question will be:

*[‘assassinated’, ‘Lincoln’]*

*Example 4.5: Keywords*

After processing the keywords will look like this:

*[‘assassin’,’lincoln’]*

*Example 4.6: Processed keywords*

The highest “matching\_sequence\_count” for any sentence can be two. It does not matter the order in which those keywords are found in a sentence, as long as they are sequential.

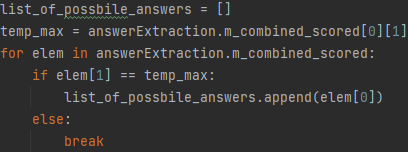
Once the matching sequence was computed for all the sentences, we calculate another metric, which is how many unique keywords can be found in said sentence. We again take every sentence and for every keyword found, a counter will be increased and said keyword will be removed from a copy of keywords to ensure that it will not be counted again. Using *Example 4.6* once again, we can see that only two keywords can be found in the query “Who assassinated Lincoln?”. This means that the maximum keyword score a sentence can have is two.

After both scores have been computed, we will use a simple addition and save the result in a list of lists called “m\_combined\_core”, and the list will be sorted in descending order by the second element. This list will look like this:

* First element is the index of the sentence.
* Second element is the combined score (longest sequence of unique keywords + numbers of unique keywords.

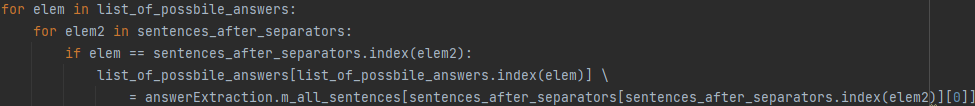
### Answer extraction

Once we have the ordered list of scores for the relevant sentences. We need to extract them and find in which one is the answer to the question. To do this we will saved the first score in the list of score (first element will always be the highest score). When we have the highest score, we iterate through the list of scores. If the score is the same as the highest one, then we will save the index of the sentence. When the score becomes less than the highest score, we stop iterating since we will never find another one that will satisfy the condition.



*Code 4.32: Saving most relevant sentence*

After having the index of the most relevant sentences, we need to get the actual sentence. To do this, we will go through “list\_of\_possible\_answers” and find the index that matches in “m\_all\_sentences”. When the index has been found, we update the “list\_of\_possible\_answers” with the actual sentence.



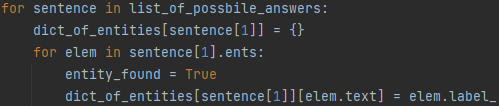
*Code 4.33: Getting the candidate sentences*

Using this list of candidates as the parameter, we will call the method “namedEntityRecognition” to get the eventual answer for the query. To find the answer we will create a list of lists called “returned\_values”. This list will store the answer to the question. Next a flag “entity\_found” is declared and set to False from the start. And lastly, a dictionary of entities called “dict\_of\_entities” is created.

Now we will go through every candidate's answer and populate the dictionary of entities in the following way. The key to the dictionary will be the sentence itself and the values will be the entities found. If only one entity is found, then the flag “entity\_found” will be set to true. The dictionary will look like this:

*{Sentence: {‘John Wilkes Booth’: ‘PERSON’, ‘David Herold’: ‘PERSON’}*

*Example 4.7: Dictionary of entities*



*Code 4.34: Creating the dictionary of entities*

Once this dictionary is created, we will check if an entity has been found. If no entity has been found in the candidate sentences, we will return the first sentence in the list of candidates. This sentence will have the highest chance of containing the answer. SpaCy is not perfect and will not recognize all entities, this is why the entity found flag has been created.

In the case where at least one entity has been found, we will create an empty dictionary called “dict\_of\_entities\_copy” and go through every entry of the dictionary of entities. A flag called “has\_prediction” will be initialized to False once we go through one key of the dictionary and a list of keys called “key\_list.” Now, we will go through every entity of a sentence and check if it matches with the prediction of the SVM. If there is a match, “has\_prediction” flag will be set to true and the entity will be stored in “key\_list”. Once all the entities from a sentence have been checked, if the flag that signals there is at least one correct entity type present is set to true, then we will populate “dict\_of\_entities\_copy” with the sentence that has relevant entities as the key and the entities are used as the values. Now we will create a new dictionary called “dict\_dependency\_entities”, and will store the following:

* The key will be the sentence itsetlf
* The values will be the entities found along with a counter that starts at 0. This counter will indicate how many keywords have a semantic relationship with the entity.

Here is an example of how will the “dict\_dependency\_entities” looks:

*{Sentence: {‘John Wilkes Booth’: 0, ‘David Herold’: 0}*

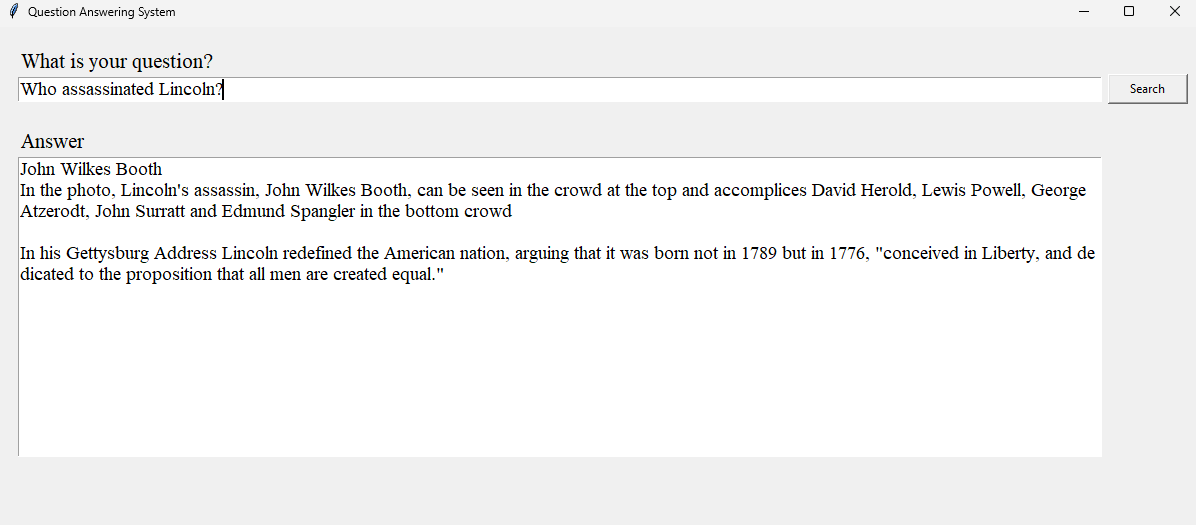
*Example 4.8: dict\_dependency\_entities example*

Now we will go through every entity and find all the words that have a semantic relationship with it. If one word is a keyword, we will increase the counter for that entity with 1. As an example, for the question “Who assassinated Lincoln?”, “dict\_dependency\_entities” will look like this:

*{Sentence: {‘John Wilkes Booth’: 1, ‘David Herold’: 0}*

*Example 4.9: Entities in relation with keywords*

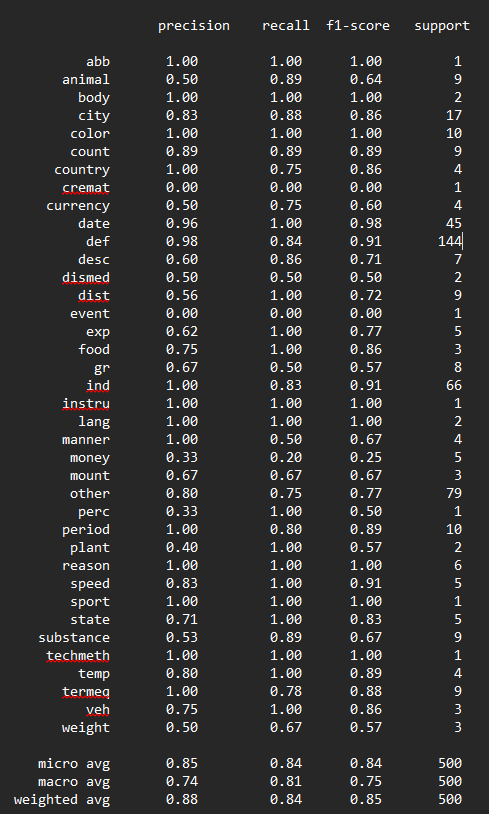
After the dictionary has been updated, we will find the highest value that is different from 0, if there is one, then we will return that entity along with the sentence where it can be found. In the case where no entity has a semantic relationship with any keywords, then we will return the first sentence since it is the most likely to have the answer for our question.



*Figure 4.4: Example of answer*

# Results

## SVM results



*Figure 5.1: SVM accuracy*

## Information retrieval



*Figure 5.2: Information retrieval accuracy*

These results have been calculated by comparing the document with the lowest Euclidian distance with the ground truth from the dataset. The accuracy would increase by taking the top 3 documents resulted but that would increase the time complexity of the program.

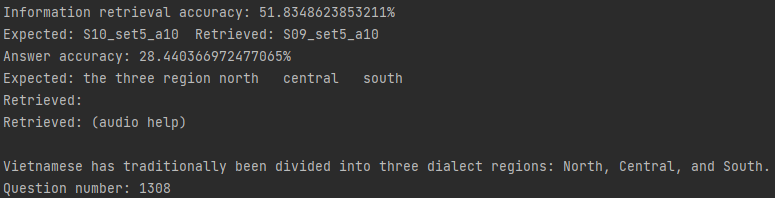
## Answering system



*Figure 5.3: Answer system accuracy*

Since answering accuracy is directly tied to the information retrieval module the results, we can consider that the system extracted the correct answer for 28% out of 51% retrievred documents. This means that the accuracy of our answer extraction module is 54%.

The final percentage is also suffering because comparing our answer with the ground truth presented in the dataset can be difficult. One example is:



*Figure 5.4: Example of correct answer categorized as incorrect*

In *Figure 5.4* we can see that our system retrieved the correct answer, but it was categorized as inccorect since not all the tokens (“the”) in the ground truth are found in the retrieved answer.

# Conclusion and further development

## Conclusion

In conclusion, all of the objectives proposed for this paper have been succsessfully implemented. Creating an SVM that can predict the type of answer needed was the first objective for the system. It was completed, and from the previous chapter we can observe that our model learnned quite well, having 84% F1-score.

The next objective was to find and retrieve the document that will most likely contain the answer for the query. This objective was completed using information retrieval techniques such as frequency vectors and Euclidian distance.

Returing the answer for a question written in natural language was an interesting objective to complete since it needed a lot of interesting techiques to achieve. Some of the most important techniques used to complete this objective were: Matching sequence score, Keywords score, Named entity recognition, and Dependency parsing.

The last objective that was completed was the creation of a GUI. This was achieved using tkinter toolkit from Python. A simple yet effective graphical interface was created using this toolkit.

## Further development

To further develop our system the following would be added:

* Our system cannot answer question that need a yes/no answer since it will return a direct answer to the question or the sentence where the answer might be found. In this case, another SVM that can recognize if a question is formulated in a way that a yes/no answer could be added along with the appropriate modules that will answer this type of question.
* Support for Romanian language: This system was designed only for English speaking users. Adding Romanian support would be a nice feature since there are not many question answering system for our language and will help the research in this domain since there are not that many papers written for Romanian language.
* Answering accuracy calculation can be increased by introducing a better comparison method between our answer and the groud truth.

# Bibliography

[1] <https://plato.stanford.edu/entries/computational-linguistics/>

[2] <https://en.wikipedia.org/wiki/Natural_language_processing>

[3] Pak, Irina, and Phoey Lee Teh. "Text segmentation techniques: a critical review." *Innovative Computing, Optimization and Its Applications: Modelling and Simulations* (2018): 167-181.

[4] <https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html>

[5] <https://en.wikipedia.org/wiki/N-gram>

[6] <https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html>

[7] <https://www.freecodecamp.org/news/an-introduction-to-part-of-speech-tagging-and-thehidden-markov-model-953d45338f24/>

[8] Brill, Eric. "A simple rule-based part of speech tagger." *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*. 1992.

[9] <https://www.mygreatlearning.com/blog/pos-tagging/>

[10] Nivre, Joakim. "Dependency grammar and dependency parsing." *MSI report* 5133.1959 (2005): 1-32.

[11] <https://www.upgrad.com/blog/dependency-parsing-in-nlp/>

[12] Mansouri, Alireza, Lilly Suriani Affendey, and Ali Mamat. "Named entity recognition approaches." *International Journal of Computer Science and Network Security* 8.2 (2008): 339-344.

[13] <https://www-nlpir.nist.gov/related_projects/muc/>

[14] Wacholder, Nina, Yael Ravin, and Misook Choi. "Disambiguation of proper names in text." *Fifth Conference on Applied Natural Language Processing*. 1997.

[15] Hearst, Marti A., et al. "Support vector machines." *IEEE Intelligent Systems and their applications* 13.4 (1998): 18-28.

[16] <https://www.tensorflow.org/datasets/catalog/trec>

[17] <https://trec.nist.gov>

[18] Smith, Noah A., Michael Heilman, and Rebecca Hwa. "Question generation as a competitive undergraduate course project." *Proceedings of the NSF Workshop on the Question Generation Shared Task and Evaluation Challenge*. Vol. 9. 2008.

[19] Lampert, Andrew. "A quick introduction to question answering." *Dated December* (2004).

[20] Moldovan, Dan, et al. "The structure and performance of an open-domain question answering system." *Proceedings of the 38th annual meeting of the Association for Computational Linguistics*. 2000.

[21] Li, Xin, and Dan Roth. "Learning question classifiers." *COLING 2002: The 19th International Conference on Computational Linguistics*. 2002.

[22] Singhal, Amit. "Modern information retrieval: A brief overview." *IEEE Data Eng. Bull.* 24.4 (2001): 35-43.

[23] W. B. Croft, L. A. Smith, and H. R. Turtle. A loosely coupled integration of a text retrieval system and an object-oriented database system. In Proc. of the 15th Intl. ACM SIGIR Conf., pages 223–232, June 1992.

[24] Rajaraman, Anand, and Jeﬀrey D. Ullman. *Mining of massive datasets*. Autoedicion, 2011.

[25] Manning, Christopher D., Prabhakar Raghavan, and Hinriche Schütze. "Xml retrieval." *Introduction to Information Retrieval* (2008).

[26] <https://en.wikipedia.org/wiki/Python_(programming_language>)

[27] <https://peps.python.org/pep-0020/>

[28] <https://www.jetbrains.com/help/pycharm/quick-start-guide.html#meet>

[29] <https://spacy.io/usage/linguistic-features>

[30] <https://www.nltk.org/howto/wordnet.html>

[31] <https://en.wikipedia.org/wiki/WordNet>

[32] <https://en.wikipedia.org/wiki/Scikit-learn>