UNIVERSITAT POLITÈCNICA DE CATALUNYA
BarcelonaTech

FACULTAT D'INFORMÀTICA DE BARCELONA

Master in Innovation and Research in Informatics
Specialization : Data Science

VIRTUAL ASSISTANT WITH NATURAL LANGUAGE PROCESSING CAPABILITIES

Applying Natural Language Processing techniques to retrieval-based question answering

CHARLINE MAS

Advisor: Lluis Padro Cirera, Department of Computer Science
Co-advisor : Imen Megdiche, ISIS Engineering school
Supervisor : Gaston Besanson, Accenture

ABSTRACT

Text processing and analysis is increasingly becoming ubiquitous due to the the immense amount of text data available on the internet. Indeed, experts have estimated that this type of data represents eighty to ninety percent of data in any organization. Therefore, techniques able to deal with unstructured data like text need to be developed.

During this thesis, an end-to-end solution was provided, creating and developing a Chatbot which, thanks to natural language processing techniques, is able to answer very complex questions, often requiring even more complex answers, in a well-defined area. To do so, Vector Space Models and Word embedding model have been studied in order to make the system understand a question and provide a pre-built answer based on the topic of the question. The best results were obtained by using the Word Mover Distance, a distance based on the Word2vec model.

The Natural Language Processing layer has been implemented into a solution composed of two user interfaces: a messaging application Telegram and a dashboard. A backend has also been designed and implemented. This project was realized entirely with Python, both the NLP study and the implementation which ended to be a reliable programming language for these kind of solutions.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Laura Cozma my mentor at Accenture, for her valuable guidance and extraordinary support in this thesis process. Furthermore, I would like to thank Dr Lluis Padro Cirera, for the useful comments, and remarks through the learning process of this master thesis. Also thanks to and Gaston Besancon, who allowed me to do my thesis at Accenture, in an innovative field: Natural Language Processing.

Their help has been essential to the completion of this work.

CONTENTS

LIST OF ABBREVIATIONS

AI: Artificial Intelligence

BOW: Bag of words representation

CBOW: Continuous bag-of-words model

DL: Deep Learning

DM: Dialogue Management

DTM: Document Term Matrix

FNN: Feed-Forward Neural Network

JSON: JavaScript Object Notation

KPI: Key Performance Indicator

LSA: Latent Semantic Analysis

LSI: Latent Semantic Indexing

ML: Machine learning

NN: Neural Networks

NLP: Natural Language Processing

PV-DBOW: Distributed Bag of Word version of Paragraph Vector.

PV-DM: Distributed Memory version of Paragraph Vector,

QA: Question answering

SIF: Smooth Inverse Frequency

TDM: Term Document Matrix

TF-IDF: Term Frequency-Inverse Document Frequency

VSM: Vector Space Model

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

1. CONTEXT OF THE PROJECT

The main goal of the project is to provide an end-to-end solution using Natural Language Processing techniques to create and develop a virtual assistant also designated as bot or Chatbot. The first part of the project consists of creating a bot able to understand, treat and answer to the user question. Then, a second part includes the creation of a system able to store the keys elements of a bot-user conversation, in order to analyzed the data and display it into a dashboard.

The use case of this project was constrained by the data. Originally, the data was supposed to be furnished by the client and be about warranty claims. However, due to the poor quality of the data and the small size of the dataset (~3k rows), this data was given up to an open source dataset that can be found at:

https://github.com/LasseRegin/medical-question-answer-data

This dataset is a questions-answers dataset. Each row is therefore, composed of a question, an answer but also a topic. While it will be described more precisely in this report, let's precise that the dataset has been reduced to the questions designated by one of the following topics: Bariatrics, Breast surgery, Cardiac Electrophysiology and Cardiology.

2. STRUCTURE OF THE PROJECT

The solution to build is composed of different elements as presented in the figure 1 below:

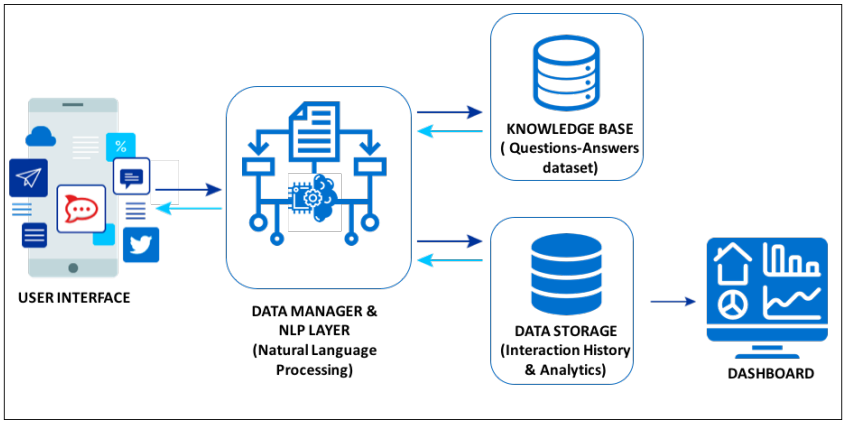


Figure 1 : Solution to be built

The different elements of the figure 1 have been divided into several sub-goals, according to the magnitude of the Master's Thesis.

1. DATA MANAGER & NLP LAYER:
   1.1. **Study and build a system able to answer very complex questions** in a well-defined domain (the Healthcare area). The focus will put on answering questions to which high-quality answers can be given by using NLP and information retrieval techniques to retrieve the answers.
   1.2. **Design a chat flow**, to keep track of the context of conversation and be able to answer correctly also called Dialogue Management (DM) system in this report.

2. USER INTERFACE & DASHBOARD:
   2.1. **Design and Implement the system into a front-end application** to make the Chatbot reachable by the users
   2.2. **Study and implement the keys KPIs** into a dashboard application

3. KNOWLEDGE BASE AND DATA STORAGE
   3.1. **Build a back-end for the system** which incorporates the system and and facilitate the creation of KPIs for the dashboard.

3. OUTLINE OF THE REPORT

In the following, the thesis is structure as follows:

- **Chapter 1: Literature Study**
  This Chapter presents a literature study into Chatbot and Natural Language processing techniques. First, the definition of Chatbot is exposed, then the various types of Chatbots are described in order to understand which type is more appropriate and why. Second, an overview of NLP techniques is given, with among them the vector space and the word embedding models that can be applied in Chatbots.

- **Chapter 2: Natural Language Processing Approaches**
  In this second chapter, the experimentation performed in order to choose the best model, is explained. First, we will described how the data has been preprocessed, then a comparison is realised between several techniques of similarity measure.

- **Chapter 3: Implementation:**
  In this chapter, the design and implementation process are presented. This part is divided into three subparts: Design process, Front-End and Backend. The Design Process section introduce the target user definition, the dashboard KPI, the chat flows along with the tool specifications. The Front-End section explains which app channel was selected and how the Chatbot and the dashboard were created. Finally, the Back-end section provides the development details, the data structure of the project, as well as the functionalities implemented.

- **Conclusion and Future Work**
  Conclusion of the project by evaluating the final solution presented and proposing future work to do.

CHAPTER 1: LITERATURE STUDY

As we consider the literature study, we note that there is a wide variety of Chatbots. In the first section we describe the different types of Chatbot. In the second section, the NLP techniques and the vector space and word embedding models used in this thesis are explained in details.

1. CHATBOTS

1.1. Definition

A Chat roBot, also known as a chatter bot, Bot or Artificial Conversational Entity, is a computer program which aims to simulate human conversation or chat, through artificial intelligence (AI). This service, powered by rules and AI, is accessible via a chat interface (User interface).

A Chabot can be used for various practical purposes from functional such as technical support to entertainment such as movie recommendation bot. They are frequently used for basic customer service and marketing systems that frequent social networking hubs and instant messaging clients such as Telegram or Messenger. They are also often included in operating systems as intelligent virtual assistants such as Cortana for Windows or Siri for Apple. Dedicated Chatbots appliances are also becoming more popular such as Amazon's Alexa. These Chatbots can perform a wide variety of actions based on user commands.

1.2. Type of Chatbot

In order to differentiate Chatbots, two axes of analysis must be taken into account:

- the type of response held by the bot
- the type of conversation.

Each of these categories is divided into two subcategories as shown in the figure 2 .

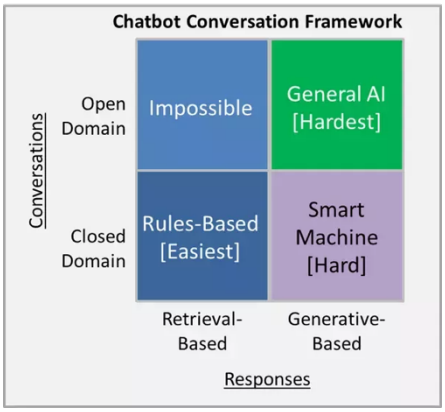


Figure 2: Chatbot Conversation Framework

1.2.1. Retrieval based models vs Generative models

The answers given by the bot can be of two types, retrieval based or generated. Bots based on the first type use a repository of predefined responses and some kind of heuristic to pick an appropriate response based on the input and context. The

heuristic depends on the complexity of the service proposed, the area in which the bot is used but could be as simple as a rule-based expression match, or as complex as an ensemble of Machine learning classifiers. One could have mentioned as downside of this type of system the limited set of possible answers that must be built upstream. However, the fact that the set of possible responses can be controlled, guarantee its quality and in some area such as Health and some use cases such as question answering, this guarantee of quality is required.

In the case of generated answers, bots are based on generative models. These systems don't reply on predefined responses; they generate new ones from scratch. Typically based on Machine Translation techniques, generative models translate an input to an output (response). This type of bot understands language not just commands, and continuously gets smarter as it learns from conversations it has with people. However, a problem with this model is the way the answers are generated. Indeed, they tends to be generic, irrelevant, inconsistent or grammatically wrong.

1.2.2. Open Domain VS Closed Domain

The conversation held by a bot can be classified as mentioned, in two categories: Open Domain and Closed Domain. In an Open Domain, the user can take the conversation anywhere. There isn't necessarily a well-defined goal or intention. This type of bot is more commonly called conversational bots.

On the other hand, in a Closed Domain, the number of possible inputs and outputs is limited because the system is trying to achieve a very specific goal. This category is often referring as task-oriented bots. As shown in the figure 1, in Open Domain conversations a bot using retrieval based model is obviously impossible to produce since imagine all the scenario of a conversation is infeasible. Using a generative model with this type of conversations, conduct to produce a general AI which aims to perform any intellectual tasks feasible by a human being. It is a primary goal of some artificial intelligence research and a common topic in science fiction and future studies. This type of AI requires a lot of means in terms of equipment (very powerful computer), data, etc.

In closed domain, on the other hand, retrieval based and generative bots (described previously) are more easily producible solutions provided you have the necessary data.

2. NATURAL LANGUAGE PROCESSING

A common approach to natural language processing is to consider it as a process of language analysis being decomposable into several stages as presented below:

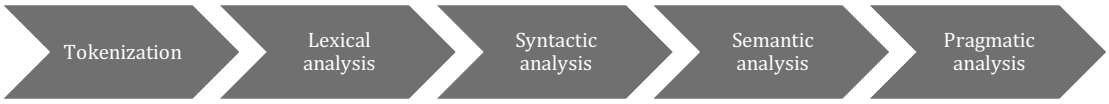


Figure 3: NLP common approach

These different stages represent different degree in the NLP analysis. Indeed, this idea is to first analyze the sentences in terms of their syntax. This analysis will provide an order and structure that is more amenable to an analysis in terms of semantics which is itself followed by a stage of pragmatic analysis. In this following, we will focus on three of five steps presented above: Tokenization, Lexical analysis and Semantic analysis. We will first start with the Tokenization, which will be include in a more general step called Text Preprocessing.

2.1. Text Preprocessing

Text Preprocessing is the task of converting a raw text file into well-defined sequence of linguistically meaningful units. It has three levels of actions: characters which represent the lowest level, words which consist of one or more characters, represent the second level. Finally, sentences which consist of one or more words, constitute the third level.

An example of actions that can be realized at the second level, is the stop-word removal. Stop-word designate the most common words in a language. For example, for the English language, the stop-words can be ‘a’, ‘the’ or ‘are’, etc. The words usually occur very frequently and cannot be used to capture the topic of one document from another. Even though the stop-word process is needed to improve the performance of text classification, it can be difficult to create standard stop-words list because of the inconsistency of words being meaningful or meaningless in some specifics domains.

Some actions can also be performed at any level, such as tokenization. Tokenization is the task of breaking down into pieces called tokens. These tokens are used for further process such as removing text component like punctuation or white space.

Text preprocessing is an essential part of any NLP system, since the characters, words, and sentences identified at this stage are the fundamental units passed to all further processing stages, from bags-of-words model through applications, such as information retrieval systems.

2.2. Lexical analysis

Text-oriented applications aim to register word's structures. To learn this representation, techniques and mechanisms of text analysis can be performed at the word's level. These kind of analysis are commonly called lexical analysis.

This type of analysis refers to the process of converting a sequence of characters in to a sequence of tokens. Thus, the focus is on the word itself and not on how it interacts with other elements of the text. The word independencies are therefore not study in this type of analysis. Some of the most important techniques of the lexical analysis are:

- **Bag-of-word (BOW)**.

A bow model is a very simplified representation of the text. In this case, the text is portrayed as the set of its words that allows multiple occurrences of the same element disregarding the grammar and even word order. This model is often used for document classification.

- **Term Frequency-Inverse Document Frequency (TF-IDF).**

TF-IDF is a numerical statistic defined by Salton and McGill in 1983. This statistic is intended to reflect how important a word is to a document in collection or corpus. The idea is, the more presents the word is in a text, the higher is the statistic. This augmentation is offset by the amount of times the word appears in all texts. The formula of the TF-IDF statistic is given below:

$$w_{t,d} = tf_{t,d} \cdot log \frac{|D|}{|d' \in D | t \in d'|}$$

In this formula,

1- $tf_{t,d}$ Represents the term frequency (TF) of term t in document.

2- $log \frac{|D|}{|d' \in D | t \in d'|}$ is the inverse document frequency (IDF), with D referring to the total number of text documents and d referring to the number of text documents, the term appears in.

With this definition, a word that appears in every document will have an almost zero IDF value, which will lead to a small TF-IDF value. However, a term that only appears in one document will have a very high IDF and thus a high TF-IDF value.

2.3. Semantic analysis

Semantic analysis describes the process of understanding natural language based on meaning and context. Indeed, in this type of analysis, structures are created to represent the meaning of words and combinations of words. In the following paragraphs we will distinguish two types of semantic analysis: Vector Space model and word embedding model.

2.3.1. Vector Space Model

A Vector Space Model (VSM) or Term Vector Model is an algebraic model for representing text documents as vectors. The VSM was developed for the SMART information retrieval system (Saltin,1971) by Gerard Salton and his colleagues (Salton, Wong & Yang, 1975). SMART pioneered many of the concepts that are used in modern search engines.

2.3.1.1. Introduction

VSM represents each documents and the user's queries as V-dimensional vectors in V-dimensional space also called document-term matrix (dtm). In this matrix, each dimension corresponds to a separate term. A term can be a word, keywords or longer phrases. In the case words are chosen to be the terms, the V-dimensional space is the size of the documents vocabulary.

$$d_j = (w_{1,j}, w_{2,j}, w_{3,j}, \dots, w_{n,j})\ q_i = (w_{1,i}, w_{2,i}, w_{3,i}, \dots, w_{n,i})$$

If a term occurs in the document, its value in the vector is non-zero. There are several ways to compute these term occurrences also known as weights. One of the best known is TF-IDF weighting as described previously. To compare the text documents vector operations that can be used. These operations are also called similarity measures. For a comparison purpose, a VSM can rank the documents based on these similarity measures.

A major limitation of the VSM is that words in one documents must exactly match words in another documents. However, this limitation can be overcome with different techniques such as word embedding described in the paragraph 2.3.2 of this chapter.

In a nutshell, a VSM can be divided into three stages. The first one is the document indexing where content bearing terms are extracted from the document text. The second stage is the weighting of the indexed terms to enhance retrieval of document relevant to the user. And finally the last stage means to ranks the documents with respect to the query according to a similarity measure.

2.3.1.2. Cosine

One of the most famous VSM based on vector operations is named cosine similarity. This measure calculates the cosine between the vectors representation of two documents using the following formula:

$$cos\ \theta\ = \frac{d_1 \cdot d_2}{\|d_1\| \cdot \|d_2\|} = \frac{\sum_{i=1}^{n} d_{1i} d_{2i}}{\sqrt{\sum_{i=1}^{n} d_1^2} \sqrt{\sum_{i=1}^{n} d_2^2}}$$

Where the numerator represents the intersection of the two document vectors and the denominator the normalization of the score by the length of the document

vectors. The normalization ensures the chance of matching independently of the length of the documents. Thus, a document with more words, will not be privilege. A high cosine value means a high similarity between two documents.

2.3.1.3. Latent Semantic Analysis

Latent Semantic Analysis originally known as Latent Semantic Indexing (LSI) is a powerful statistical technique. LSA is based on two main steps. The first one concerned the construction of a term-document matrix (TDM) M. The size of M is $n * m$ where the rows correspond to m terms, the columns correspond to n documents and $M_{[i,j]}$ corresponds to the frequency of the term i in the document j. The second step is the singular value decomposition where the TDM M will be decomposed into three matrices as follow:

$$M = U * S * V^{\top}$$

U and $\mathrm{V}^{\top}$ are two orthogonal matrices and S which is a diagonal matrix. SVD is a matrix algebra technique which essentially re-orients and ranks the dimensions in a vector space. Because the dimensions in a vector space computed by SVD are ordered from most to least significant, if some of the less significant dimensions are ignored, the reduced representation is guaranteed to be the best possible for that dimensionality. Finally, based on the equation below, only the k largest singular values and their corresponding singular vectors from U and $\mathrm{V}^{\top}$ will be used in order to reduce the semantic space which corresponds to M_k:

$$M_k = U_k * S_k * V_k^t$$

2.3.2. Word embedding

2.3.2.1. A brief history

The term word embedding was originally coined by *Bengio* et al. in 2003 who trained them in a neural language model together with the model's parameters. In 2008, *Collobert and Weston* with their paper *A unified architecture for natural language processing*, were arguably the first to demonstrate the power of pre-trained word embedding. But it was 2013, *Mikolov et al* who really brought word embedding to the fore through the creation of Word2vec, a toolkit that allows the seamless training and use of pre-trained word embedding, signaling that word embedding had reached the mainstream.

The aim of word embedding is to build a low dimensional vector representation of word from a corpus of text. One of the main advantage of word embedding is a more expressive and efficient representation maintained by the contextual similarity of words and a low dimensional vector. Word2vec, one of the most famous word embedding algorithm will be presented in the paragraph 2.3.2.3. But before, we will in the following paragraph briefly described the Neural Network algorithm.

2.3.2.2. Neural Network (NN)

Human brains are composed of billions of cells, working together, called neurons. These neurons can solve complicated problems fast. The idea of the method is to imitate the properties observed in biological neural systems through mathematical models. This system is named neural networks.

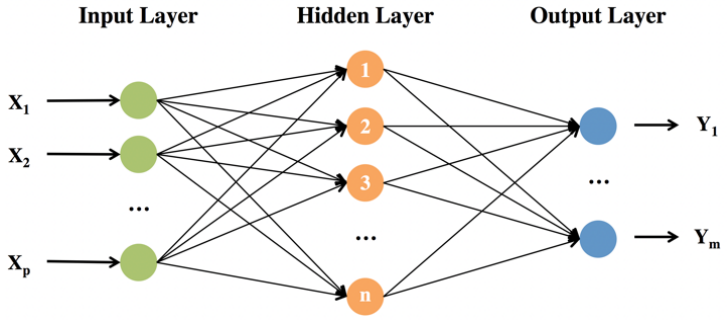


Figure 4: NN Structure with one hidden layer

A neural network can be represented as shown in the figure 4, where every node represents a neuron. Each of these nodes are modulated by their corresponding weights and applies a certain activation function over its input to determine its outputs. An artificial neural network can consist of many more layers than in the figure 4, and all layers between the input and output layer are called hidden layers. The input layer has as many neurons as there are independent variables, and the output layer has as many neurons as there are dependent variables. The amount of hidden layers, and the amount of neurons in the hidden layers depends on the type and amount of data.

Usual activation functions are:

- Identity: $\sigma(s) = s$
- Threshold: $\sigma(s) = \begin{cases} 1, if\ s > 0 \\ 0, if\ s = 0 \\ -1, if\ s < 0 \end{cases}$
- Logistic: $\sigma(s) = \frac{1}{1+e^{-s}}$
- Hyperbolic Tangent: $\sigma(s) = \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}}$
- Gaussian Radial Basis Function: $\sigma(s) = e^{-\frac{1}{2}s^2}$

During the training of neural networks, the weights on the connection links between the neurons are modified, as to reach the optimal model for the training dataset an element of feedback is therefore required. This element is called backpropagation. It distributes the error term back up through the layers. The larger the difference between the model outcome and the actual outcome, the more the

connection weights will be altered. Once the network is done training, it can be presented with new inputs to generate responses.

The network in figure 4 is a feed-forward neural network (FNN) as the information propagates only in one direction i.e. wherein connections between the nodes do not form a cycle as opposed to Recurrent Neural networks that can learn, thanks to bi-directional data propagation, the vector representations from words and can remember a huge context.

2.3.2.3. Word2vec

Word2vec is one of the most popular word embedding model. Indeed, it is a computationally-efficient predictive model for learning word embedding from row text. The main principle of this method is to learn law dimensional vectors from the begging.

Two architectures are proposed for learning word embedding: the Continuous Bag-Of-Words model (CBOW) and the Skip-Gram model. These models are algorithmically similar, except that CBOW predicts target words from the surrounding words, while the Skip-Gram does the inverse and predicts the context from the center word.

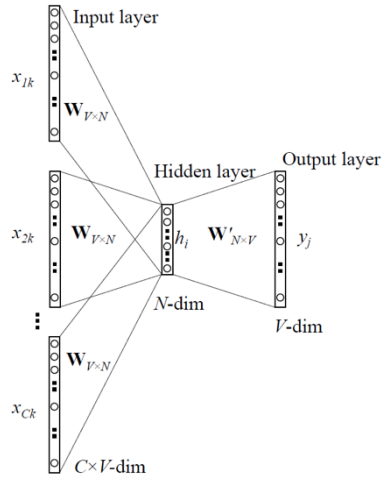


Figure 5: Word2vec, CBOW model

More precisely, CBOW corresponds to the neighboring words in the window. In this neural network presented in the figure 5, three layers are used.

First, the input layer corresponds to the context, then the hidden layer corresponds to the projection of each word from the input layer into the weight matrix which is projected into the third layer names output layer. The last step is then, the comparison between the model output and the word itself to correct its representation based on the back propagation of the error gradient. Therefore, the purpose of CBOW neural network is to maximize the equation presented below, where V is the vocabulary size, c corresponds to the window size of each word.

$$max \sum_{t=1}^{V} log\ p(m_t | m_{t-\frac{\epsilon}{2}}, \dots m_{t+\frac{\epsilon}{2}})$$

Skip-gram is the opposite as shown in figure 5, since the input layer corresponds this time to the target word and the output layer corresponds to the context. Therefore, in this algorithm the last step consists of the comparison between its output and each word of the context. Here also, this comparison aims to correct the representation obtained, based on the back propagation of the error gradient.

In this case, the maximization seeking is presented below, with V corresponding to vocabulary size and c corresponding to the window size of each word.

$$max \frac{1}{V} \sum_{t=1}^{V} \sum_{j=t-c, j \neq t}^{t+c} log\, p(m_j | m_t)$$

The major limit of these models comes from the learning of the output vectors which can be a difficult and expensive task. To address this problem, two algorithms can be used: Negative Sampling algorithm and Hierarchical softmax.

Negative Sampling algorithm aims to limit the number of output vectors that need to be updating. Thus, only a sample of the output vectors is updated based on a noise distribution. This distribution is a probabilistic distribution which is used in the sampling process. Hierarchical Softmax, on the other hand, is based on Huffman tree, a binary tree resenting all terms based on their frequencies. In this tree, each step from the root to the target is normalized. It is the training data which determines which algorithms works better.

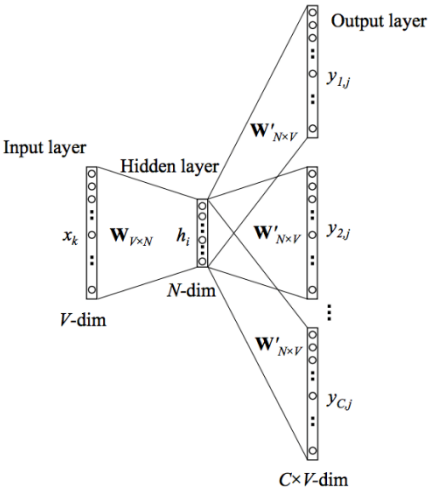


Figure 6: Word2vec, Skip-gram model

2.3.3. Similarity measures based on Word2vec

2.3.3.1. Doc2vec

Doc2vec is an implementation of paragraph vectors, a generalization of Word2vec to documents. This implementation was made by the authors of the Gensim Python library, much use in the field of NLP. The goal of doc2vec is to create a numeric representation of a document, regardless of its length. However unlike words, documents do not come in logical structures, therefore another method had to be found.

In 2014, Le & Mikolov proposed a method that learns fixed length feature representations for various length texts called Paragraph2vec. This is achieved by training a small neural network to perform prediction task. The labels required to train such a model, are coming from the text itself. In the architecture of Paragraph2vec, the input contains a vector that represents the document as shown in the figure 7 or in the figure 8.

While many other methods can be used to represent sentences, paragraphs or documents as a fixed size vector, Paragraph2vec yield vectors of a more manageable

size. As for the Word2vec model, Paragraph2vec has two ways of being computed: PV-DM and PV-DBOW.

The PV-DM stands for Distributed Memory version of Paragraph Vector and is an extension of the CBOW model but instead of using just words to predict the next word, a feature vector is added as shown in the figure 7. This vector is document-unique. In the CBOW of Word2vec, the model learns to predict a center word based on the context. Similarly, in PV-DM, the main idea is: randomly sample consecutive words from a paragraph and predict a center word from the randomly sampled set of words by taking as input the embedding words and a paragraph id. The figure 7 presents the following elements: Paragraph Matrix, Average/Concatenate and Classifier.

- The Paragraph Matrix, is a matrix D where each column represents the vector of a paragraph.
- Average/Concatenate signify that the word vectors and paragraph vector are averaged/concatenated.
- Classifier, averaged/concatenated hidden layer vector as input and predicts the center word.

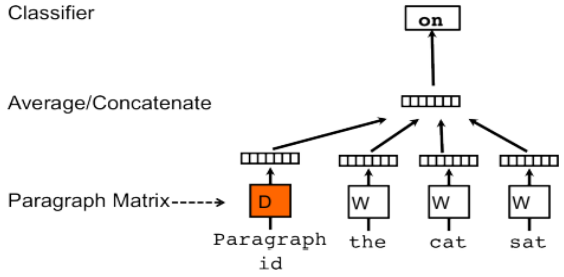


Figure 7: Distributed Memory version of Paragraph Vector

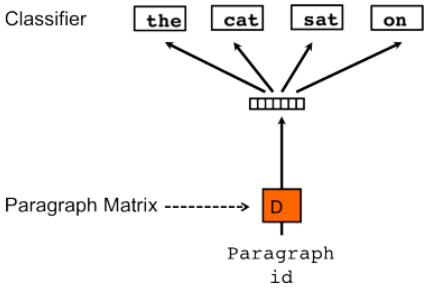


Figure 8:Distributed Bag of Word version of Paragraph Vector

The PV-DM acts as a memory that remembers what is missing from the context (as the topic of the paragraph) thanks to the document's vector. Thus, the word vectors represent the concept of a word and the document vector intends to represent the concept of a document.

As mentioned, there is another algorithm, which is similar to Skip-Gram named PV-DBOW for Distributed Bag of Word version of Paragraph Vector. This model, shows in the figure 8, is different since it ignores the context words in the input and is forced to predict words randomly sampled from the paragraph in the output. In this case, the algorithm is faster and consumes less memory, since there is no need to save the word vectors and has less parameters that need to be trained.

2.3.3.2. Word Mover Distance

The Word Mover's Distance (WMD) is claimed to achieve better results than other baselines when used to compute document similarity. WMD uses the word embedding of the words in two texts to measure the minimum distance that the words in one text need to "travel" in semantic space to reach the words in the other text. More precisely, this distance is based on the Earth Mover Distance, and addresses the transportation problem by measuring the distance between two distributions in some regions, where the pairwise distance between points in the ground distance.

In 2015, Kusner et al. proposed a novel distance function between text documents called Word Mover's Distance (WMD). The Word Mover Distance is viewed as an instance of the Earth Mover Distance. The figure 9 illustrates the concepts of the Word Mover's Distance, where the semantic space is learned by the Word2Vec model.

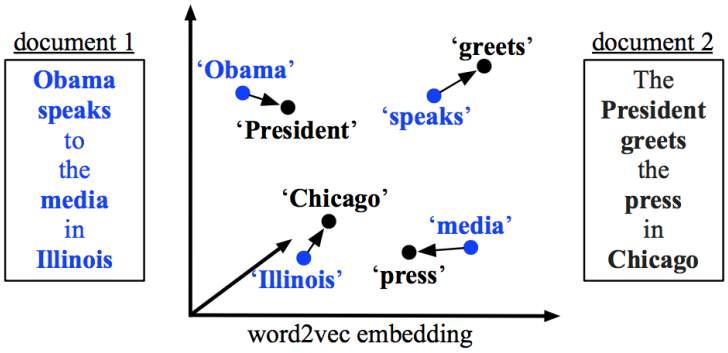


Figure 9: Words distance in the Word2vec semantic space

In the WMD computation, the dissimilarity between two words as a natural building block is assumed to create a distance between two documents. To estimate the distance between pairs of words, the embedding that are learned with the Word2vec are used. The word travel cost or words dissimilarities are provided by Euclidean distance between words in the Word2vec embedding space:

$$c(w_i, w_j) = \|V(w_i) - V(w_j)\|$$

where, w_i and w_j are two words, while $V(w_i)$ and $V(w_j)$ are their word embeddings. The distance or travel cost between two documents is defined as the minimum (weighted) cumulative cost required to move all words from document $d_1$to document d_2:

$$C_{wmd}(d_1, d_2) = \min_{F>0} \sum_{wi \in d_1} \sum_{wj \in d_2} F_{wiw_j}\, c(, w_i, w_j)$$

Subject to: $\sum_{w_j} F_{w_i w_j} = d_{1i}$ and $\sum_{w_i} F_{w_i w_j} = d_{2j}$

Where F is a flow matrix indicating how much of w_i in document d_1 travels to w_j in document d_2. The notation d_{1i} is the document representation by normalized BOW (nBOW), for example, word i appear c_i times in the document, it denotes:

$$d_i = \frac{c_i}{\sum_{j=1}^{n} c_j}$$

CHAPTER 2: NLP APPROACHES

This chapter presents the study realized to accomplish the objective 1.1. of this project: *Study and build a system able to answer very complex questions*. First, the problem definition is described, then the experimental approach is presented.

1. PROBLEM DEFINITION

One of the goal of this project is to incorporate into the final system a dialogue management system and a NLP layer, in order to distinguish different types of input and treat them according to the context of the conversation. Thus, several types of input will be considered for this system and described in the chapter three of this report. For this study, we will only focus on one type of input and therefore considered that the DM system already selected the incoming message as being what we will call a *Health question* (also designated as query).

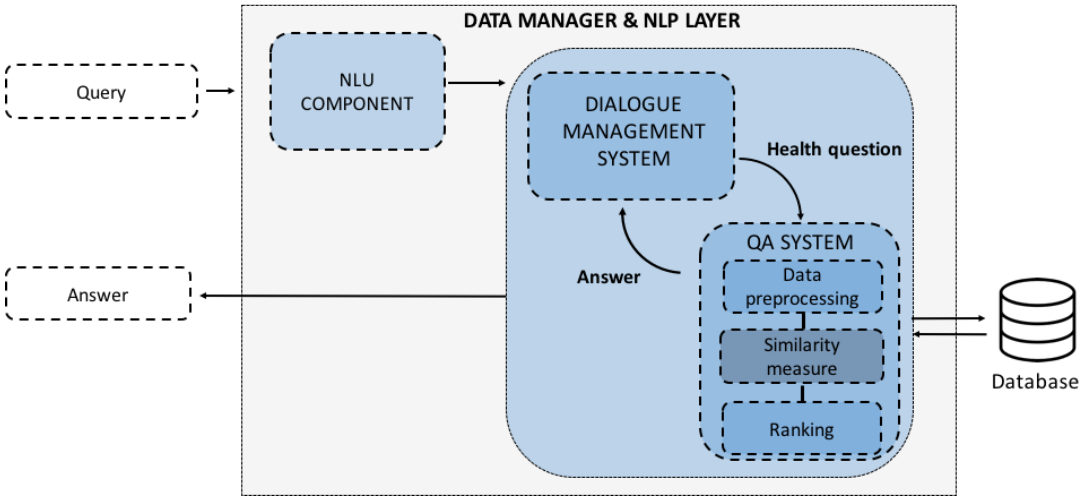


Figure 10: Data manager and NLP layer of the system

As shown in the figure 10, a Health question is, once being detected, sent to a Question-Answering (QA) system. Indeed, since the answers given by the bot will come from a Questions-Answers dataset, the idea is to build a QA system which aims to retrieve the “best answer” from the dataset.

Is designated as “best answer”, a message answering in the most meaningful way to the query. This answer is linked to the question designated as the most similar to the query sent by the user. Since the data is composed of question-answer pairs, once the most similar question is found, retrieve the best answer is easy. The challenge is therefore, to find a model able to identify the most similar question.

A QA system is based on similarity measures. The main idea is to compute the similarity between a query and all the questions of the dataset. Then, the obtained similarities measures are ranked, as shown in the figure 7, from the highest to the smallest knowing that the higher the similarities, the better.

Textual data can be tricky to analyze depending on the context in which the text has been written. For example, a Wikipedia article is long and well written while discussion on Forums are mostly based on short and often not grammatically correct Question-Answers. The challenge is therefore, to understand what the user asked. Or in other words, to obtain words vectors describing in the most meaningful way in which context appears the word.

In order to find the most appropriate similarity measure for the QA system, the similarity methods presented in the chapter 1, will be study:

- Latent Semantic Indexing (LSI) weighted by TFIDF
- Cosine similarity weighted by TFIDF
- Doc2vec
- Word Mover's Distance.

2. EXPERIMENTAL APPROCHE

In this part, the model described in the previous chapter, will be evaluated on the data of the project. The methodology used for this experiments is shown in the figure 11. First the data will be collected, cleaned and explored. Then, the similarity models will be computed. Lastly, the comparison of the different methods will be based on their retrieval task.

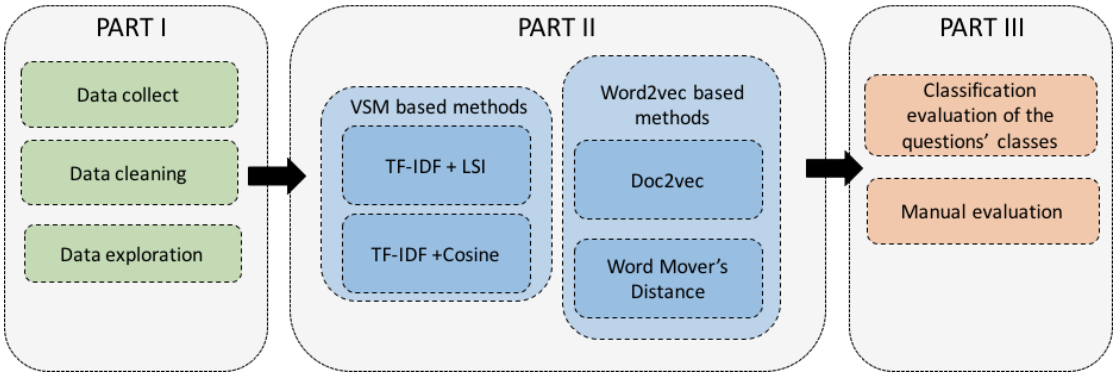


Figure 11: Methodology process

2.1. Data

As mentioned in the introduction, the data used for this project can be found on GitHub. The dataset is open source and composed of question-answer pairs sorted by topics which have been gathered from several health forums on 5th of May 2017:

- Ehealthforum: https://ehealthforum.com/
- Healthtap: www.healthtap.com/
- Icliniq: https://www.icliniq.com/
- questionDoctor: https://questiondoctors.com/
- webmd: www.webmd.com

The data is stored in six JSON files in which the following elements can be found:

- **Question**. The question asked by a user on the forum
- **Answer**. An answer gave by a health professional as pretended on the different forums. (For this project, we assume that is true)
- **Tags**. One or several topic(s) of the question (and answer).
- **URL**. Link where can be found the discussion.

A fifth element can be found in the files, however, being different for each file, this element will not be considered in the rest of this experiment. Same with the URL element that we will not considered for the following. In total, the data contains at the begin of the study, **166804** rows i.e. question-answer pairs.

2.1.1. Data cleaning

A first glance at the data reveals that there are many unanswered questions as well as duplicates question-answer-topic set. Since the final model will be used to return answers from the dataset as an answer, the set of answers needs to be as correct as possible. Therefore, the dataset is cleaned by removing question-answer pairs that do not meet our criteria.

Second, after removing the rows having an empty answer, a checking is realized to visualize the number of topic per question. Following this analysis, the graph

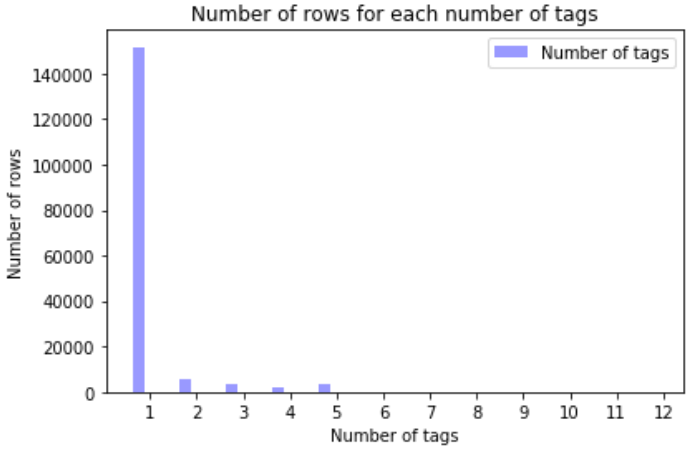


Figure 12: A bar chart displaying number of question per number of topics (tags)

shown figure 12 was obtained. It appears that the number of questions with a single tag represents 90% of the data.

Therefore, it was decided to only keep the questions marked with one topic. Among these selected questions 95826 were remaining. However most of them were appearing only in one or two questions, therefore it was decided to select the topics present in a majority of questions, the most 6 frequents are presented below:

TOPIC	NUMBER OF QUESTIONS
BARIATRICS	30516
CARDIAC ELECTROPHYSIOLOGY	21666
CARDIOLOGY	17761
BREAST SURGERY	11142
WOUND CARE	4136
PREGNANCY	348

Table 1: Top 6 most frequents topics

Since the number of questions with the topic pregnancy and wound care are really low compare to the first four, these topics and all the topics having a lower frequency were discarded. This reduced the dataset to **81040 rows**. The topic remaining were the following:

- **Bariatrics**. Branch of medicine that deals with the causes, prevention and treatment of obesity.
- **Breast surgery.** Form of surgery performed on the breast which can included Breast reduction surgery, Augmentation mammoplasty, Mastectomy, Lumpectomy, Breast-conserving surgery (a less radical cancer surgery than mastectomy), Mastopexy, or breast lift surgery, Surgery for breast abscess, (including incision and drainage as well as excision of lactiferous ducts), Surgical breast biopsy and Microdochectomy (removal of a lactiferous duct)
- **Cardiac Electrophysiology.** Science of elucidating, diagnosing and treating the electrical activities of the heart.
- **Cardiology.** Branch of medicine dealing with disorders of the heart as well as parts of the circulatory system.

After, this first cleaning some observations were made:

- There is a very large variation in question and answer length
- There are many spelling mistakes in the questions
- Some answers contain hyper-links

- Answers are formulated for specific cases
- Some questions and some answers contain a salutation

A lot of sentences (questions and answers) contain abbreviations such as “er” standing for “Emergency room” or “ekg” for “electrocardiogram”. Most of these abbreviations were health related.

2.1.2. Natural Language Processing cleaning

In this section NLP techniques are applied to clean deeper the data. Indeed, the questions need to be prepared for modelling while the answers need to be prepared to be send to the final user. Although the answers were written by health professional, there are some elements such as misspelled words or white space that need to be removed.

As mentioned, some questions and answers contain noisy elements such as salutations. To improve the dataset from which we return answers, we first remove these noisy elements. Then, we processed the questions through a NLP pipeline to prepare them for the modelling part.

2.1.2.1. Removing salutations from questions and answers

In the dataset can be found, some salutations and closings words such as “Hello” or “Hi” or “Thank you” etc. It was decided to remove them for two reasons: first, the salutations in the answers will confuse the user since this type of answers is given in the middle of the Chat flow (presented in the next chapter) and therefore, the conversation will have started already. Second, this type of words will bring noise into our models.

The removal of these elements was done with the pseudo-code in the following algorithm:

Algorithm 1: Replacement of words quoted as greetings

1: Define *greetings_list*
2: Break up each question into its sentences
3: Collect “greetings words”: for each sentence of the question,
 3.1: Detect via *regex*[1] if one of the word of the sentence is in *greetings_list*
 3.2: If one or more words are detected, remove it/them.
4: Do the same for answers.

[1] Regex: A regular expression is a special sequence of characters that helps to match or find other words or sets of words, using a specialized syntax held in a pattern.

2.1.2.2. *NLP cleaning pipeline*

The first cleaning phase revealed key points that needed to be take into account. Therefore, the following NLP pipeline was built as shown below:

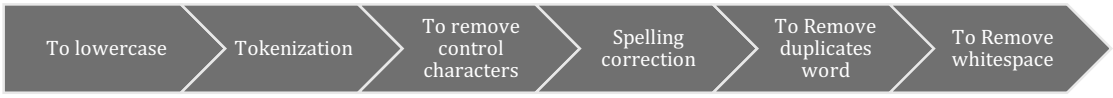


Figure 13: NLP cleaning pipeline

First each sentence went through the *tolower* python function. This step refers to the lower-casing NLP technique. All the letters of the dataset are changed to lower-case to make sure that identical words match each other, regardless of the letters begin lower-case or upper-case.

Then, each sentence of the questions column and the answers column of the dataset are tokenized with the NLTK python's package and more precisely the *word_tokenize* function. This tokenization, explained in the chapter 1 part 2.1, was applied by separating all words and punctuation with comma.

The third step is where the control characters were remove thanks to the regex functions (see Footnote in the previous page).

The step four is taking care of the spelling correction. The spelling mistakes can influence matching a new question with the questions in the dataset. Therefore, a spelling corrector was applied based on the algorithm 2 presented below.

Algorithm 2: Spelling correction

1: Calculate the probability of this word occurring for this specific dataset. This is done by dividing the frequency with which the word occurs in the dataset and dividing it by the total number of words in the dataset.

2: Find all edits that are only one correction away from the word. One edit can be a deletion of a letter, an insertion of a letter, a swap of two adjacent letters or a replacement of one letter for another.

3: Find all edits that are two corrections away from the word. This means running step 2 again on its own outcome.

4: Restrict the sets of words (one and two edits away from the initial word) to only include words that are in our existing word list.

5: Generate the corrected spelling candidates for the word. These candidates are (in order of replacement): the word itself (if it is in the word list), words in the word list that are one edit away, words in the word list that are two edits away, and finally if none of these exist it will be the word itself.

6: Correct the word. This is done based on the word candidates generated in the previous step. If the word itself is in the word list it will not be altered. If the word is not in the word list, the first option is the words that are one edit away and in the word list. If these exist we choose the one with the highest probability of occurring in our dataset, as calculated in step 1. Otherwise, we look at the words that are two edits away and in the word list, again choosing the one with the highest probability

of occurring. If all of these options do not generate words, we just give the word that was supplied back as this is then a rare word that is potentially important for the meaning of the question

In step five, we remove duplicates words present in the same sentence, next to each other such as:

*"your doctor can **tell tell** you if you have thyroid problems"* → *"your doctor can **tell** you if you have thyroid problems".*

The step six was a checking and removal of the white spaces which correspond to two characters.

2.1.2.3. Stop-words removal

Finally, a last step was performed on the questions. Indeed, while they are now cleaned, some noisy elements are remaining and will keep our models to perform efficiently. Since the questions of the dataset won't be seen by the final user, all the stop-words, punctuation and other elements qualified as noise such as numbers or hyper-links can be removed. Although essential for reading, these elements greatly disrupt the models and must be deleted. The logic to remove the stop-words is exposed in the algorithm below:

Algorithm 4 Stop-word removal

1: Remove all standard English words using a pre-made English-word list.
2: Remove punctuation
4: Remove all words which contain numbers
5: Remove all hyper-links.
6: Change all punctuation that is still present (this is only possible inside words), to spaces. For example, "high-glycaemic" is changed to the words "hyper" and "glycaemic".

2.1.3. Situation after the cleaning

After applying the cleaning step, the data has been reduced by 47% which leads to a dataset with **81040** rows. A new column has been added to the dataset names *cleaned_questions.* This column represents the specific cleaning apply in the second phase of the cleaning. The table 2 below shows an example of row of the dataset.

Question	Answer	Tags	Cleaned_questions
When dieting how can you avoid waking up throughout the night really hungry?	Dieting does not mean going hungry. Dinner should have enough proteins healthy fats and some carbs with plenty of veg.	bariatrics	['dieting', 'avoid', 'waking', 'throughout', 'night', 'really', 'hungry']

Table 2: Example of the cleaned dataset

The *cleaned_questions* column is going to be used to train the models. Therefore, it is interesting to see the number of common words that can be found between the different topics as shown in the table 3 below. For example, the topic *Bariactrics* has 5106 words that also appear in some of the question of the *Breast Surgery* topic. This means that the words represent a risk to be misinterpreted by our models and attribute to a *Breast Surgery* question instead of a Bariatrics one.

	Bariatrics	Breast Surgery	Cardiac Electrophysiology	Cardiology
Bariatrics	-	**5106**	**7240**	**6309**
Breast Surgery		-	**4657**	**4224**
Cardiac Electrophysiology			-	**6446**
Cardiology				-

Table 3: Common words between two topics

A checking was also realized a checking to see the number of common word to all the topic: **3488** words were found**.** The graph figure 14 shows the most 10 frequents words of these common words.

The important numbers of shared words show that some words appear a lot in the dataset. Since they can designate several topics, we will consider two case to treat them in the following section: Penalize these words via TF-IDF method or rely on the Word2vec training to learn correctly the context in which these words are appearing.

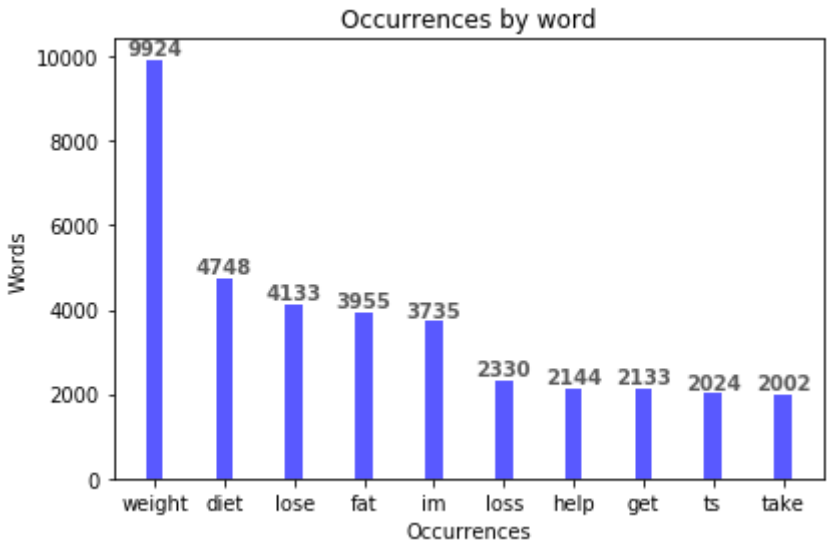


Figure 14: Top 10 most frequent words in all the dataset.

2.2. Models comparison

Now that we have clarified the pre-process step, in this section we are going to focus on the models presented in section 2.3 of the chapter 1 and study which one is the more appropriate for our project. First, we will explain the evaluation process that has been chosen and then we will present the results.

2.2.1. Evaluation methods

To check the performance of a model, we used a train-test methodology, being:

- **Training Set.** A fraction of the entire dataset was used for training purpose.
- **Testing Set**. The model trained with the Training Set was tested with the Testing Set. Thereby, we checked the model performance with observations that were not used for the training.

A common proportion used to divide a dataset into a train and test is 80-20, with 80 % of the data going to the training set and 20% to the test set. However, the WMD is very slow to be computed. Therefore, the test set will only be composed of 1000 questions picked randomly.

Models will be evaluated on their effectiveness of estimating similarities between a question of the test and all the questions of the dataset. The evaluation metric is the accuracy obtained. It is computed by comparing the topic of the question detected by the model, to the test set query. Therefore, the accuracy will represent the number of good topic detected. The algorithm used is shown below:

Algorithm 5: Evaluation models

1: For all the queries of the test set:
1.1: Compute the distance between the query and all the questions of the dataset
1.2: Ranked the similarities obtained from the highest to the smallest
1.3: Select the highest similarity
1.4: Retrieve the index of the question with the highest similarity
and its tag from the dataset
1.5: Save the elements retrieved
2: Compute accuracy

A second step, performed in this study, is the evaluation of the quality of the answers. It consists of manually checking the results. Indeed, we chose to perform a manual evaluation for the following reasons: First, our dataset is composed of 81040 rows with a lot of questions for each topic. Manually paired questions, even a few hundreds to allow us to have a more reproducible experiment would have taken a lot of time and since the project was constrained by the magnitude of the thesis, it appeared to be infeasible. Second, given the nature of the problem, (i.e. pairing questions about one of the topic presented in the previous sections), there is not a single answer (i.e. a single question-question pair) that could have been created. Therefore, just paring one expected question could have been count as wrong if the model had brought a similar and valid but not paired question.

For these reasons and because it was too long to check the 1000 questions retrieved by each model, it was decided to manually check 300 good classified questions. The idea was to check if the questions designated as most similar are meaningful.

2.2.2. Accuracy evaluation

In this section, we present the first experiments realized to compute the accuracy for each model.

2.2.2.1. Cosine and LSI

In the paragraph 2.1.3. of this chapter we saw that some words are present in a lot of questions independently of the topic. Therefore, it was decided to penalized these words with the TF-IDF method explained in the section 2.3.1 of the first chapter.

For this computation we used *TfidfVectorizer* from the Python package Sklearn. Basically, every time we have a new query, a document-term-matrix is built, combining the query and all the questions of the dataset.

Then on one hand, we apply on the dtm matrix obtained the cosine similarity between the first row of the matrix (which is the pre-processed query) to the others.

On the other hand, the LSI model was computed with the Python package Gensim and more specifically the module concerning the Latent Semantic Analysis (aka Latent Semantic Indexing).

The result obtained are shown in the table below:

	Cosine	**LSI**
Accuracy	81%	80%
Average computation time for each query	2,3 secs	3,5 secs
Total amount of computation time	39 min	50 min

Table 4: Cosine and LSI accuracy results

From the table 4 presented above, we can see that globally our models performed quite well since they both have about 80% of accuracy. However, in term of time the Cosine computation is faster.

2.2.2.2. Word Mover's Distance

For this experiment, a python version of the WMD, from the Gensim package has been used. In our implementation, word distance was estimated by the Euclidean Distance calculated against the Word2vec vector space. We first learned a Word2Vec model on word vector dimensions' equals 100, and for a question from the test, we computed its Word Mover Distance with all the question of the dataset.

The computation of the WMD is quite long, about 2min to compute the distance query-all questions of the dataset. Therefore, calculate this distance for all the question of the test set, is in this condition infeasible. However, one way to proceed is to use the Prefetch and prune algorithm introduced in [2], it is a relaxation of the distance computation problem to prune documents that are not in the number n nearest neighbors. Therefore, the computation of the true WMD is not done for these documents. In this experiment, we compute the WMD distance for 20 nearest neighbors.

	WMD with skip-gram	WMD with CBOW
Accuracy	87%	86.2%
Average computation time for each query	4.124 secs	4.126 secs
Total amount of computation time	34 h	34h

Table 5: CBOW and Skip-Gram models accuracy results

From the table 5, it seems that the model computed with the Skip-gram algorithm performed slightly better than the one with CBOW. The computation time for the two models, is quite long. Indeed, it took to 34h in average to compute the distance for the 1000 questions.

Due to the very expensive computation memory and the time required to compute these distances, it was impossible to run several times the experience to determine the best parameters for the word2vec models.

2.2.2.3. Doc2vec

To compute the Doc2vec model, we used here also, the Gensim library. Doc2vec is, as explained in the chapter one, a Paragraph2vec implementation in Python. First a comparison analysis has been made with little tuning, so to make a preliminary filtering:

- Window Size = 8
- Min_count = 2 (filters out words with frequency less than 2),
- Iteration = 20.
- dbow_words=1 (trains word-vectors (in skip-gram fashion) simultaneous with DBOW doc-vector training)
- vector_size = 100 (Dimensionality of the feature vectors.)

We compute both type of Doc2vec model (PV-DM & PV-DBOW). For each vector size, after the learning process, pairwise distances between posting vectors are estimated by the Cosine Distance. The results obtained with the default parameters are presented in the table 6.

	PV-DM	**PV-DBOW**
Accuracy	**33.5%**	**65.0%**
Average computation time for each query	**0,01 secs**	**0,01 secs**
Total amount of computation time	**10 secs**	**10 secs**

Table 6: PV-DM and PV-DBOW accuracy results

From the table above, we can see that the PV-DBOW perform way better than the PV-DM model. Therefore, we will for the following tests, only keep the PV-DBOW model.

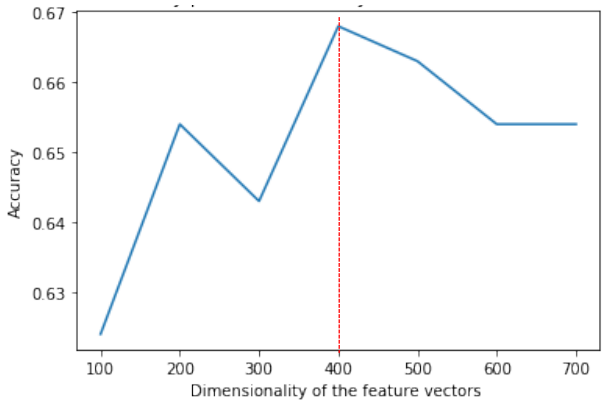


Figure 15: Accuracy per Dimensionality of the feature vectors

Since the computation of the Doc2vec models were fast (about 10 secs to compute all the distances), we were able to train different paragraph vector sizes: 100, 200, 300, up to 700. We could not train vectors with a dimensionality higher than 700 because we had insufficient memory for the computation.

The results, obtained after the computation of the several models and represented figure 15, show an improvement of the accuracy when the dimension of the paragraph vector is a bit increased until the dimension 400 (note that the y axis is from 0.63 to 0.67 to ensure a better visibility). Therefore, we will keep the size of 400 to compute the models in the next experimentation.

Another evaluation was made on the alpha parameter which represents the learning rate. To determine the optimal value on alpha, the model was trained on several values of alpha from 0.005 to 0.0095. The results are presented on the graph present in the figure 16.

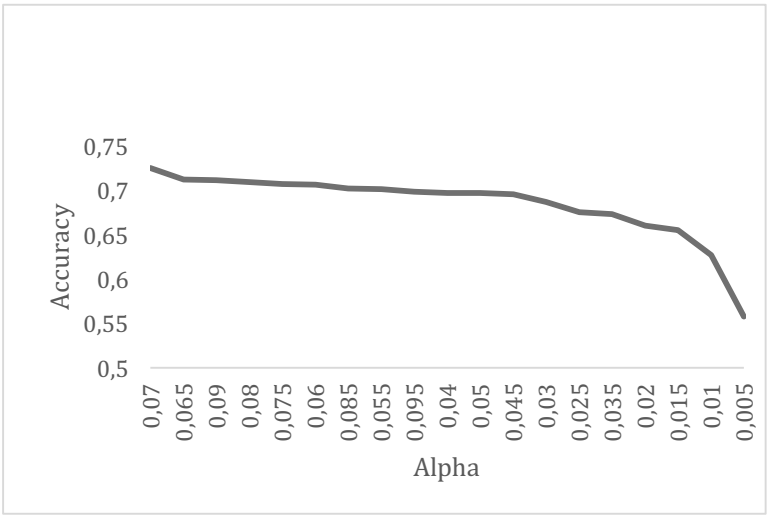


Figure 16: Accuracy according to the alpha value

Here also the accuracy doesn't move a lot, however the best result is obtained with an alpha equal to 0.07. The final accuracy obtained was **73%**.

2.3. Result summary and manual evaluation

Finally, the different accuracies and times computation for the models are resumed in the table below:

	Cosine	LSI	WMD Skip-gram	PV-DBOW
Accuracy	81%	80%	87%	65.0%
Average computation time for each query	2,3 secs	3,5 secs	4.124 secs	0,01 secs
Total amount of computation time	39 min	50 min	34 h	10 secs

Figure 17: Sum up of all the results

We got high accuracies especially from the Word Mover Distance, computed with the skip-gram algorithm. However, like we said these results need to be checked to be sure that the answers gotten by the models are meaningful.

Therefore, as explained in the section 2.2.1 of this chapter, a manual evaluation was realized. This evaluation has been made through the following process:

1- Retrieve the first 300 good classified questions for each model
2- Evaluate them
   - If the question retrieved means the same as the original question, the score is put to 1. Else, it marks as 0.
   - Compute the total score for each model

An example of the manual evaluation is given below:

Query	Questions retrieved by skip_gram_word2vec	score
Should i take blood pressure medication to lower my blood pressure during the pregnancy?	*What can i take to lower my blood pressure?*	1
What are the consequences of forgetting diabetic gh blood pressure meds?	*What are the consequences of gh and low blood pressure?*	0
When doing blood pressure what is meant by systolic and diastolic?	*What are the differences between systolic and diastolic blood pressure?*	1

Table 7: Example of the manual scoring

The results for this manual scoring evaluation are presented in the table below:

	Cosine	LSI	WMD Skip-gram	PV-DBOW
Score	69**%**	67 %	77 %	38%

Table 8: Manual evaluation results

After the manual evaluation, from the results show in the table 8, we can say that the Word Mover distance appears to get the better results. Between Cosine and LSI, we can say that there is not a big difference. However, concerning the PV-DBOW model, clearly does not retrieved a lot of meaningful questions.

However, it is important to also take into consideration the computation time of each of the model. Technically, it is the doc2vec model which performed the best but since it got the lowest accuracy and the lowest score this model was discard. The second methods which can be computed quite fast is the cosine similarity. This method got correct result to the manual evaluation and got an accuracy of 81 %, however it is still far from the score obtained by the word mover distance.

In the end, it was decided to privilege the quality of the questions retrieved by the bot to the detriment of time. The WMD distance was therefore chosen to be implemented into the system to treat the input of type *Health question.*

Other types of input will be treated by our bot such as *questions definitions* or *already asked questions*. We will explain in details all the types and their meaning in the part 1.3.1. of the next chapter, however for some of the inputs, similarity measures will need to be computed. Since the Word Mover Distance rely on the Word2vec model and therefore require a minimum amount of data to be train, it was decided to use the second best model, i.e. cosine similarity, for all the inputs for which we don't have a lot of data to compare with.

CHAPTER 3: IMPLEMENTATION

In this chapter three different part are presented. First, the design process of the solution is explained. Then, the front-end implementation is presented and the background development is showed.

1. DESIGN PROCESS

While the NLP part represents the core of a Chatbot. Many things must be taken into account when implementing it. Indeed, it is important to define the kind of public that will use the tool in order to create interactions as simple as possible and user friendly. But also, to build an efficient database to easily collect and retrieve the data for further analysis. Thus, this chapter will present the design process of the Chatbot.

1.1. Target group

The tool to develop has two different target users:

1. **Bot's users**

The bot's users are in this project, the main target. Indeed, this concerns all the people who will ask questions and discuss with the bot. The messages coming from this type of users should be: greetings messages to start the conversation, answer to the bot questions and Health question(s) about one of the topics presented in the previous chapter, part 2.1.1.

In this category, we can also distinguish two subcategories: Unknown and known user. An Unknown user, is a user who is using the bot for the first time while a known user already used the bot at least once. It was decided that only user knowing the password could use the bot, therefore an unknown will be asked to give a password to start talking with the bot.

2. **Professionals**

The "Professionals" type of users, represents the client or the entity for which the bot is working. This type of user is not talking to the bot, but should be able to access to the data gathered by the system. Therefore, the device expected for this user is a dashboard in which, the data will be analyzed through several KPIs (Key Performance Indicators) to provide relevant information on the data collected.

1.2. Dashboard KPI

As mentioned before, this project target two types of users, one of them is called *Professionals*. It was decided that this type of user would have access to a dashboard to measure the evolution of the bot. Thus, some metrics to compute and visualize the performance needs to be defined. There are two types of information that we are interested in:

- The traffic evolution
- The content of the messages exchange with the bot, also named trends

Therefore, the dashboard will be a trade-off of operational dashboards, which focused on the traffic evolution and Analytical dashboards, which process data to identify trends. In order to design the several KPI, we based our reflexion on the AAARR start-up metrics model developed by Dave Mcclure which was partially adapted. Thus, the following KPIs were selected and divided into three categories:

- **Activation Rate**: Designate all the metric related to traffic information which includes:
  - Total number of users
  - New users
  - User average age, to estimate the age range of the users
  - User session location, to identify in which country/city the bot is used
  - User global satisfaction
- **User Interactions:** Gather, the metrics to evaluate the interaction between the bot and the users.
  - The average holding time, indicating the average time of a discussion between the bot and the user
  - The average number of question per users
  - The total number of questions
- **Analytics:** To analyze the user's questions and the trends**.**
  - Trend topic
  - Most frequent words in user's questions
  - Most frequent asked questions

Some KPIs such as *the user average age* or *the user session location* require information that cannot be retrieved from the data gathered by the system but that need to be ask to user. Therefore, a conversation needs to be built in order to bring the user to give us this data. Beside, to make our bot looking more human, it needs to speak like it. In the following section, the chat flows built for this bot will be explained.

1.3. Chat flows

In this section will be described how a bot's conversation is built. Indeed, to ensure the consistency in the responses sent to the user and to gather the required information, the bot needs to be able to keep track of the context of the current discussion and to ask the right question at the right moment.

To help the bot to keep track of this context, several *States* need to be defined. A *State* represents a part of the conversation and indicates the type of action to be performed. For example, the state *START* refers to the begin of a discussion between the bot and the user. Thus, the bot is expecting a certain type of messages such as "Hi" or "Hello" and will perform the action *sentMessage,* to reply to the user with a welcome message. Therefore, the different type of message that could be received by the bot will be presented first and then the different *States* used by the bot will be explained.

1.3.1. Message types

In this project, several categories of messages have been identified as possible type of input for our bot as presented below:

- **Greetings.** Includes all messages such as "Hi", "Hello", "Good morning", etc.
- **User personal information.** As seen in the previous part, several personal information from user are required to display the KPIs. The expected data are the birth date, the current city and country of the user. The user's location information could be recovered via the phone's location; however, the bot will also be accessible via a web interface so it was decided to directly ask the user.
- **Topic related.** Includes the topic selected and all messages indicating the topic select is not the topic desired.
- **Question**. Includes, the questions about the definitions of the topics such as "What does Bariatrics mean?", these types of questions will be called *Questions definitions*. Another type is the questions about Health such as the ones with have in the dataset. These questions will be called *Health question.* Finally, a last type of questions that can be identifying by the bot: the *already asked question* which designate a question that have been asked before by the user as much in terms of meaning (question with the same idea but asked with different words) as in terms of words ((question asked with the same words).
- **Satisfaction**. Boolean message: Yes/No
- **Bye**. Includes messages such as "Bye", "Good bye".
- **Command.** Two command are used for this project: */start* and */done*.
- **Password**. An unknown user must know the password in order to converse with the bot.

1.3.2. Conversation states

As mentioned previously, a conversation is divided into states which work as gates. Once the user gave the type of answer waited, the state is changed for another one and so on. If the user has not answered the question in the expected way, the state won't change. In this project, four main *States* were defined:

1. START.
2. LOCATION.
3. IDENTIFICATION.
4. FALLBACK.

In the following section will be described the operation of each state.

1.3.2.1. START

The state START is responsible of checking the user identity as shown in the figure 18. Three types of scenarios can be seen in this state. First, the user is known. In this case, a welcome message will be sent and the state will be changed to IDENTIFICATION. Second, the user is unknown, and the message received is the password. In this case, User personal information are asked to the user and the state is changed to LOCATION. Finally, if the user is unknown and the message is of greetings type, the password is asked and the state remained unchanged.

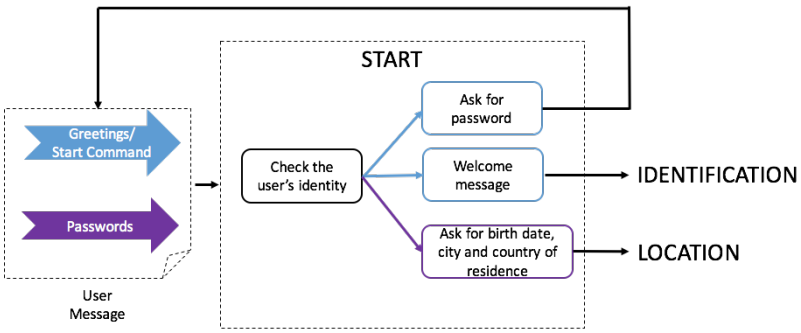


Figure 18: State START

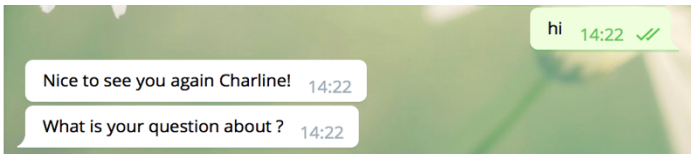


Figure 19: Example of a conversation in the START state, with known user and greetings input

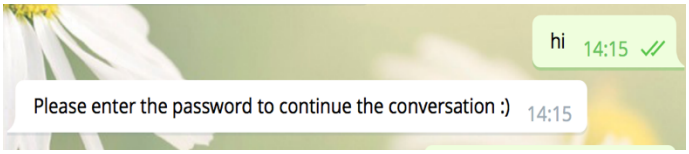


Figure 20: Example of a conversation in the START state, with greetings message as input and unknown user

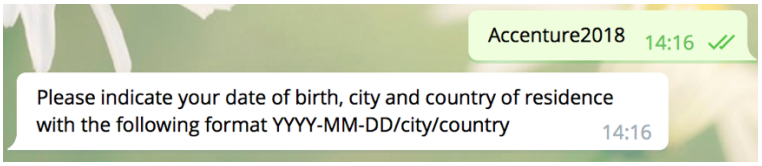
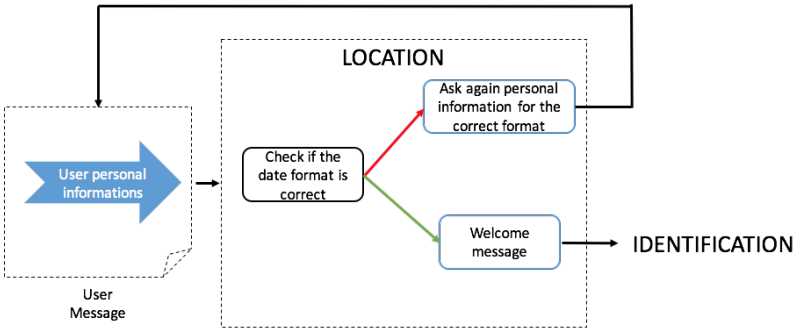


Figure 21: Example of a conversation in the START state, with password as input and unknown user

1.3.2.2. LOCATION

The state LOCATION aims to check the format of the personal information given by the user. Thus, two scenarios can happen. First, if the birth date has been given with the correct format, a welcome message is sent and the state is modified to IDENTIFICATION. Second, the date format was incorrect. In this case, the user personal information is asked again with a reminder on the expected format.



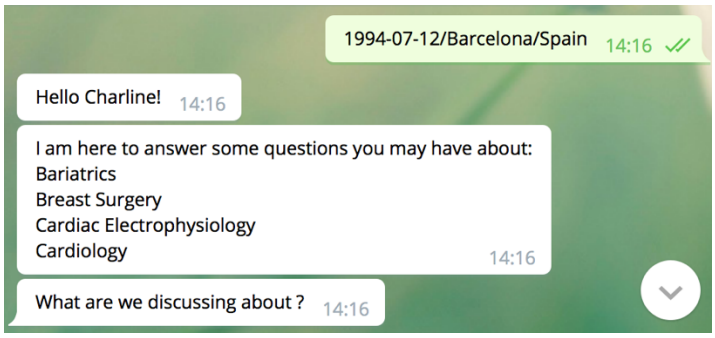


Figure 22: Example of LOCATION state, with correct format message

1.3.2.3. IDENTIFICATION

The state IDENTIFICATION is the main the state of the conversation. Indeed, once the conversation is in this phase, the following types of messages can be treated: *Topic related*, *Question,* and *Satisfaction*. This state is therefore, in charge of analyzing the incoming message and detect in which category it belongs to. Several scenarios can happen here.

1.3.2.3.1. Topic related messages

First, the incoming is identifying as *Topic related.* In this case the message could either be the name of one of the topics or a sentence indicating that the chosen topic is actually **not** the desired one. First, If the input is a topic's name then a global variable will be implemented in background with the name of the topic and the user is asked to ask his question. Second, a message indicating that a wrong topic has been chosen, will lead the bot the ask again the topic desired.

In the case, the global variable is not filled but the detected input is *Question* or *Satisfaction* the user will be asked to choose a topic.

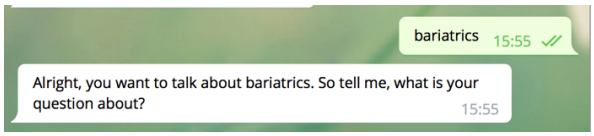


Figure 23: Example of a input topic selected

1.3.2.3.2. Questions

Second, A *Question* type input is identified and the global variable concerning the chosen topic is filled. In this case, the input is pre-processed using the same NLP pipeline as explained in the section 2.1.2 of the chapter 2, the similarity between the pre-processed incoming message and some predefined questions are evaluated with the cosine similarity measure. If the measure is above the threshold, then the question is identified as a *Question definition* and an answer is sent to the user. On the other hand, if the question is not identified as *Question definition* then cosine similarity is applied again but on all the questions previously asked the user. If the user is of type Unknown or if the user did not ask a threshold number of question, this step is skipped. After measure of the similarities between the old user questions and the new query, if the highest similarity found is above a threshold then the question is identify as *already asked question,* otherwise the question is considered as *Health Question* and will be analyzed via *Word Mover's distance (WMD).* Again, if the measure is above the threshold then an answer is sent. Else, the bot is sending an apology and asks the user to rephrase his question.

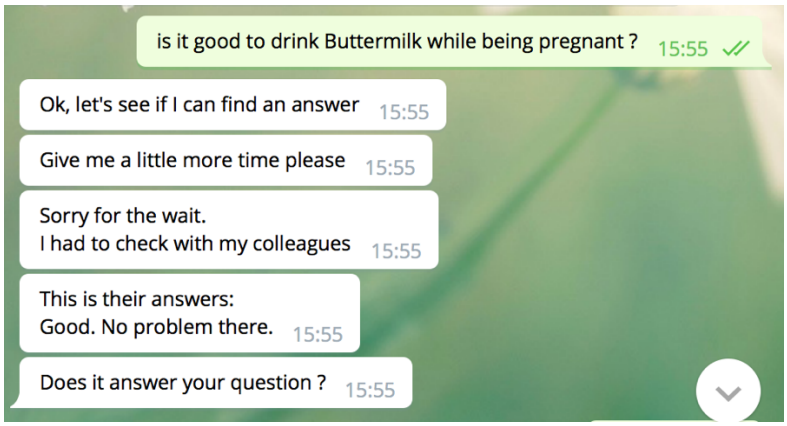


Figure 24: Example of IDENTIFICATION state, with Health questions *type identified*

In the case of a *Health questions* type, we can see in the figure 24 that several messages are sent before, sending the actual answer. It is explained by the very long computation time required by the wmd. Indeed, this messages are here to make the use wait and leave the application.

1.3.2.3.3. Satisfactions

Third, a Yes/No message is identified. If the input is a *Yes*, then a *Thank you* message is sent. But if it is a No, then the bot we will send the answer of the second most similar question identified by the WMD and so on. The number of No is recorded and if this number reaches the threshold then an apology is sent to the user with indications about how to contact a professional.

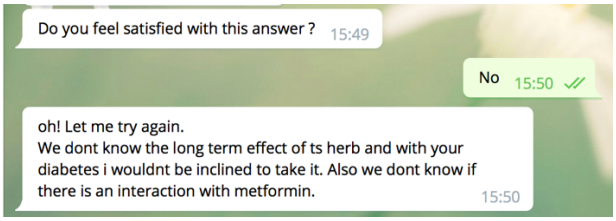


Figure 25: Example of SATISFACTION state with No as input

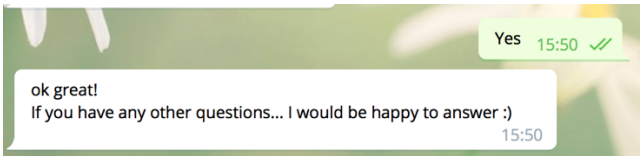


Figure 26: Example of the IDENTIFICATION state with Yes as input

1.3.2.4. FALLBACK

The FALLBACK state is a closing state. It is going to end the conversation when the incoming message is of type *Bye*. In this case, only one scenario is possible: a *Bye* message is sent to the user and the state is changed to START.

The states *START* and *FALLBACK* represent the begin and end of the chat flow, which means that a conversation will always begin in the *START* state and end in the start *FALLBACK.* This implies that every time a user is going to send a greeting message, the conversation will be placed in the state *START*, even if the current the state was *LOCATION* or *IDENTIFICATION.* Same with the *FALLBACK* states, a *bye* message will always end the conversation. This was made to make the bot looking more like a Human. Indeed, if for example, the user leaves the chat without sending a *bye* message. The state of the conversation is therefore still *IDENTIFICATION*. In this case if the user come back, the possible inputs would have been *Topic related*, *Question* or *Satisfaction*. Or when you chat with a person, the first thing you say before starting a new conversation is “Hi” or “Hello”.

1.4. Programming Language

Now that the key elements have been defined for each interface, a programming language needs to be selected for the backend and for the connection between the backend and the bot interface. For creating Chatbots, there are various options to choose in programming language. Some of the most popular languages to build Chatbots are presented below:

- **Python**. Well known for its simplicity, Python has a straightforward syntax and it's object-oriented. This language is one of the most widely used in programming languages in the field of Artificial Intelligence.
- **Java**. Provides all the high-level features needed in AI project. Java has the most important features for a sophisticated interface, like facilitated visualization and standard Widget toolkit.
- **Ruby**. Very simple syntax which allows beginners to create a Chatbot easily. It is a dynamic and object-oriented language.
- **Javascript**. High-level interpreted programming language. This language is widely in AI platforms. It supports real-time messages and is easy to learn.

For this project a simple, flexible and easy language is sought. Since the NLP part is the core of the Chatbots, a programming language with a NLP functions is required. Therefore, **Python** was chosen for the following reasons:

- **Easy to use**: Python is to read which make it easy to pass the project from one colleague to another
- **Productivity**: It is a great language for building scalable multiprotocol network applications. Therefore, it is very suitable for building Chatbots.
- **Machine Learning and Deep Learning Framework:** Python has wide array of open-source libraries including Scikit-learn and Tensorflow. It also includes state-of-the-art AI algorithms.
- **NLP libraries:** It is the most popular language for natural language processing and the biggest community. Indeed, one of the reason is Natural Language Toolkit (NLTK) which was developed for Python and is one of the best framework for text mining.

2. FRONT-ENDS

This part will focus on the realization of the two interfaces: Chatbot and the dashboard.

2.1. Chatbot interface

Messaging platforms are becoming universal mobile apps. Since, businesses look for a way to deliver their messages and services where the consumers are, chat platforms are becoming more and more popular for the companies and Chatbots give them a way to do this. For this a project, it was decided to use an existent messaging application and the following aspects were taken into account when selecting the app:

- **Device Support:** The app must be available on mobile and computer.
- **Channel Usability**: The priority will be given to the most commonly used channels, in order to give more visibility and easier access to the Chabot.
- **Development Flexibility:** The channel must allow the most flexible way of development.
- **Security**: The chosen channel, must make sure that communication between two parties cannot be intercepted, altered, forged, or read by unauthorized third parties.

2.1.1. Chatbot channel

There are several channel commonly used as presented in the list below:

- **Facebook Messenger.** Facebook, one the most popular social networks, possesses one of the larger amount of user with different age range.

- **Skype and Skype business.** With the same idea as Facebook, Skype bots are generally used in group chats for functional and business purposes.

- **Telegram.** Telegram is a multi-platform instant messaging created by Pavel Durov. It is unique in its openness, as it has open-source client applications and an open, flexible protocol. However, Telegram is mostly popular for its focus on the users privacy, point that most of the other instant messaging clients put aside.
- **Slack**. Slack is an instant messaging client and foremost a workplace communication tool. However, it can also be used for customer support, online communities and in some cases even communication between social groups from the real world.

In this project, the Chatbot aims to answer patient's questions, therefore a one-to-one, private conversation is considered. This first criterion discards the channels Slack, Skype and Skype Business, since the group oriented is not wanted here. This lead to a comparison of the Messenger and the Telegram channels.

At the end, it was decided to use Telegram as the front-end up for the following reasons:

- Telegram Messenger is accessible from multiple devices (mobile, computer, tablet) and platforms (Android, iPhone, iPad, Microsoft Windows, Web-version, macOS, PC, Mac, Linux), which makes it very reachable.
- Although it has less active users than Facebook Messenger, it still has got 100 million daily active users and it is growing at a rate of more than 50% annually.
- Finally, and the most decisive factor, is its flexibility. The Telegram Bot has got multiple functionalities that enriches the user experience, provides a huge range of possibilities when developing a Chatbot and allows us to implement the NLP layer built in the chapter two.
-

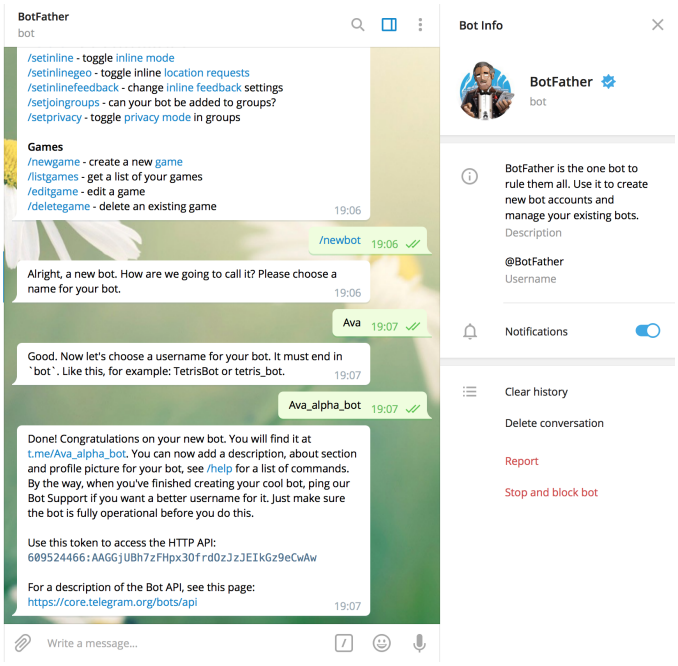
2.1.2. Chatbot registration

Now that the channel has been chosen, the bot must be created on it. In fact, to allow the bot to be seen and reachable on Telegram it needs to be register. Telegram provides a very simple way to create bots, indeed after a simple conversation with the bot called BotFather, the bot can be registered.

The registration process required two elements:

Bot's Name. The name of the bot will be displayed in the contact details and elsewhere. In this project, it was decided to name the bot **Ava**.

Bot's Username. The Username is a short name, to be used in mentions and telegram links. Usernames are 5-32 characters long and are case insensitive, but may only include Latin characters, numbers, and underscores. It is also important to notice that the username must end in 'bot'. For this project the usernames used is **Ava_alpha_bot.**



At the end of this procedure a token is delivered. A token is a string along the lines of *110201543:AAHdqTcvCH1vGWJxfSeofSAs0K5PALDsaw* that is required to authorize the bot and send requests to the Bot API. The token delivered for Ava bot is the following ***609524466:AAGGjUBh7zFHpx3OfrdOzJzJEIkGz9eCwAw.***

2.1.3. Telegram bot API

As described on the Telegram Bot API Web Documentation, the Bot API is an HTTP-based interface created for developer keen on building bots for Telegram. Therefore, a bot is controlled by HTTP requests to the Bot API indicating the Bot's token as follow: *https://api.telegram.org/bot<token>/METHOD_NAME.* In the following section will be described: API requests operations and the python wrapper used.

We will describe how works the API requests and then explained the Python wrapper used in this project.

2.1.3.1. Telegram bot API

Two ways of receiving updated can be used: The *Polling* method or the Webhooks as shown in the figure 27.

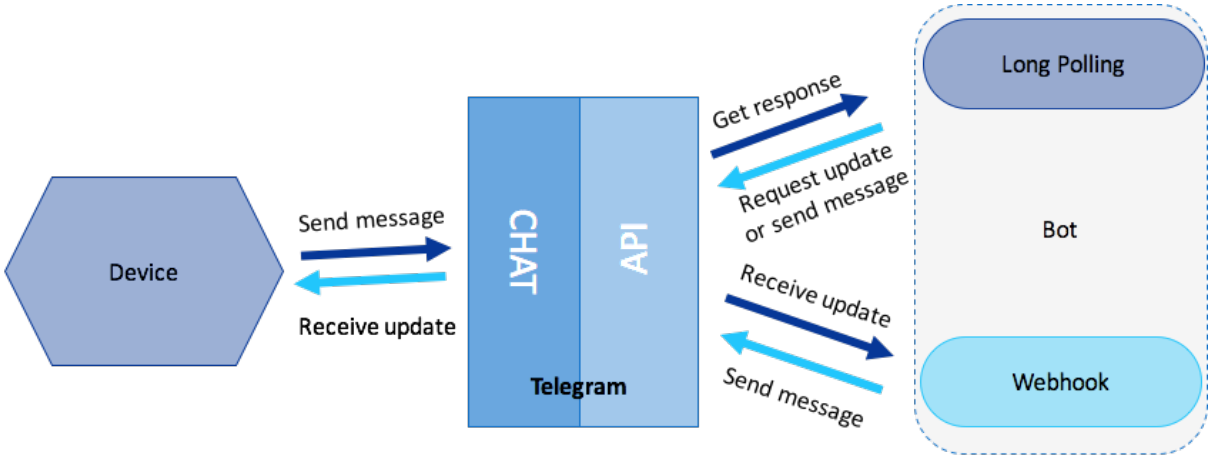


Figure 27: Long Polling and Webhook methods

- The ***Long Polling*** through the *getUpdates* method is sending an HTTPS GET request to the API. *Long Polling* allows the connection to stay open until updates are received. This connection is ruled by a timeout argument which defines the time that can stay a connection open.
- The ***Webhook*** method requires to specify the URL and the incoming updates received via an outgoing webhook. Every time there is an update for the Bot, the Telegram API will send an HTTPS POST request to the specified URL, containing the update.

Regardless of the chosen option, the update comes as a JSON-serialized objects. These objects contain the information about the messages sent by the user to the bot. Incoming updates are stored for 24 hours on the server.

In this project, the *Long Polling* method was used as it was easier to set up with the chosen programming language.

2.1.3.2. Python wrapper

To manage the Bot with python, the *python-telegram-bot* library was used, and more precisely the submodule *telegram.ext*. This library provides a python interface for Telegram Bot API and will take care to send and get message from the Bot API thanks to the token.

Several classes can be found in this package, among them Updater, Dispatcher, ConversationHandler, CommandHandler, MessageHandler and Filters. These classes have used in the project and are described below:

- ***Updater***. The purpose of the *Updater* is to receive updates from Telegram and give it to the *Dispatcher.* This class contain the *getUpdate* described in the previous paragraph.
- ***Dispatcher***. This class dispatches all kinds of updates to its registered handlers. In this project, the updates will be send to the *ConversationHandler*.
- ***Handlers***. The *Handlers* aim to handle the different type of messages that the bot will have to deal with. Among them, we used *ConversationHandler*, *CommandHandler*, *MessageHandler* .
    - ***ConversationHandler****.* This class is charge to keep track of the context of the discussion with the different states described in the paragraph 1.3.2 of this chapter.
    - ***CommandHandler****.* This class will handle the command message, i.e. the message written after the slash symbol such as */start*.
    - ***MessageHandler****.* This class will handle all the non-command messages.
- ***Filters****.* This class filter out the non-command messages and specified type of text. It is used with *MessageHandler* , to defined the input that will accept the class.

A documentation of this library is also available at https://python-telegram-bot.readthedocs.io/en/stable/telegram.html

2.2. Dashboard interface

Now that the bot's user front-end has been defined, the professionals interface needs to be built. In the first part of this chapter, the KPIs for this dashboard have been selected. In this paragraph, we will first focus on the software used to build our dashboard and then, a presentation of the final interface will be done.

2.2.1. Dashboard software

Nowadays, modern dashboards use data visualization to improve the user experience of traditional business intelligence. Data visualization is one of the most popular business intelligence tools. Indeed, it helps people to effectively see and understand data. As the industry continues to grow, so does the push for design-focused, thoughtful, user-friendly dashboards. In order to choose, the more appropriate dashboard software for this project, a comparison was made based on the following criteria:

- Free Trial
- Free Version Available
- Automated Visualizations
- Visualization Option / User Palette
- Customizable Dashboards
- Sharing / Publish Tool
- Community Marketplace / Gallery

To realize the comparison, the most common dashboards software were selected and presented below:

- **Microsoft Power Bi:**
  A extremely powerful platform with a lot of data source connectors, a user-friendly interface and good data visualization capabilities.
- **Tableau**:
  A good platform with a lot of data connectors and visualizations. The design is here also very user-friendly. This product has large community of user.
- **Google Analytics:**
  Due to its brand recognition and the fact that it's free, Google Analytics is the biggest name website and mobile app intelligence.
- **Charito:**
  Processing engine with a powerful analytics platform, which possesses a good query optimization system on SQL. It is entirely web-based.

	Microsoft Power Bi	Tableau	Google Analytics	Charito
Free Trial	Yes	Yes	Yes	Yes
Free Version Available	Yes	No	Yes	No
Automated Visualizations	Yes	Yes	No	Yes
Visualization Option / User Palette	Yes	Yes	Yes	Yes
Customizable Dashboards	Yes	Yes	Yes	Yes
Sharing / Publish Tool	Yes	Yes	Yes	Yes

Table 9: Dashboard software comparison

The table 9 presents the comparison realized between the different dashboard software. The one which corresponds the most to the criteria is the software **Microsoft Power Bi**.

2.2.2. Final dashboard

The final dashboard is composed of two pages:

- Global indicators
- Bot Analytics

The Global indicators page is designed to ensure an overall view of the general metrics gather from the bot database. Thus, are present in this first page the following indicators:

1. The total number of users of the application
2. The new users having been registered during the current month
3. The average age of the users
4. The total number of questions
5. The average number of question per users
6. The user sessions location, showing the location of each users.
7. The average holding time, indicating the average time of a conversation with the bot
8. The average abandoned rate, evaluating the number of user having been registered but didn't ask any questions.
9. The global satisfaction, showing the general user's satisfaction
10. The membership evolution (per month), which plot the evolution of the number of new user per month.

A filter was added to this page, in order to filter by time, the following data displayed.

The Bot Analytics page, also having a time filter, is designed to allow an analytic study of the questions asked by the users. Thus, are present in this first page the following indicators:

1. The trend topic, indicating the most popular question's topic asked per month
2. The latest questions asked
3. The top 10 words occurrence, which show the keywords usually mentioned in the questions
4. The words location (per country), indicating where the question(s) having the selected keyword, were asked.
5. The words cloud, showing the top 50 most frequent words

3. BACK-END CHATBOT

3.1. Data structure

In this part, the Data structure of the project will be presented.

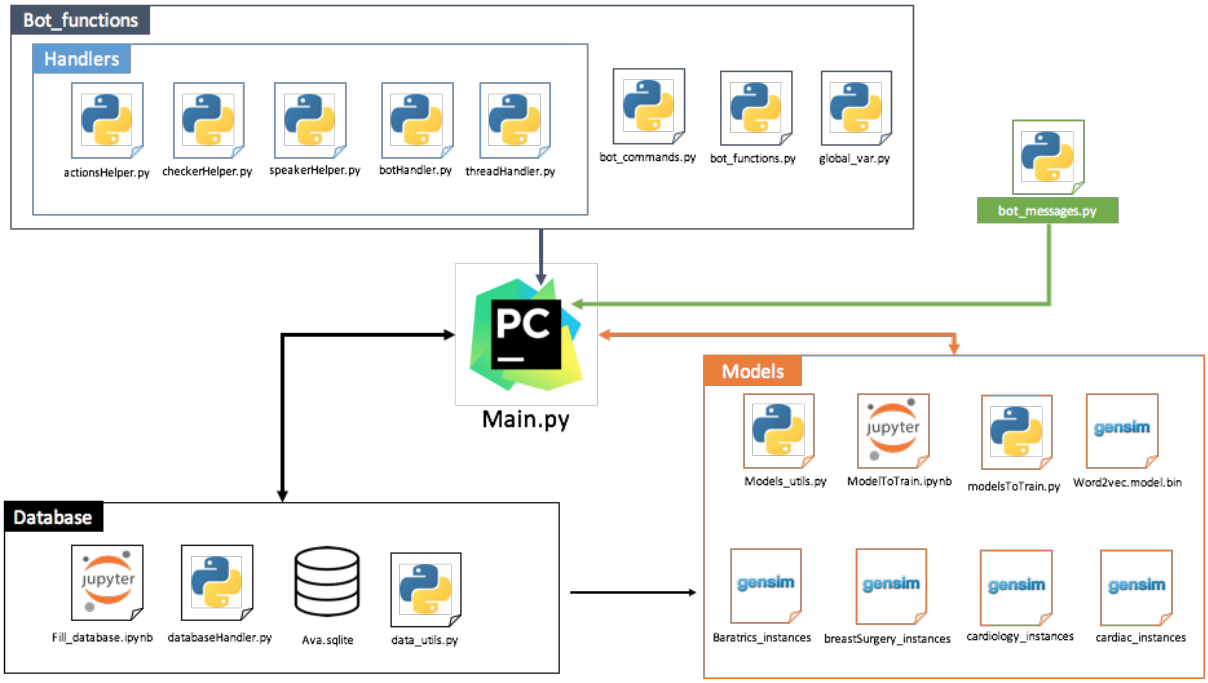


Figure 28: Data structure of the project

There are three folders in this python project:

- Bot functions: All the scripts responsible of the communication with the user and the Telegram API.
- Database: All the files related to the data management and storage.
- Models: Contains the scripts related to the training, models computations and models files.

In the following sections, all the files will be briefly described.

3.1.1. Main.py file

This script is core of the Chatbot. Indeed, all the functions, objects and data are loaded into this files. The bot object and the conversation object are created here. The conversation object is composed of three elements:

- Entry-points: It is the state *START* presented in paragraph 1.3.2 of this chapter. The entry_points is a python list, which can be composed of one or many elements. In this project a *CommandHandler* is added in order to allow the input */start*. A *RegexHandler* is added, which allow the state to accept the message

of type *greetings.* If the input match one of the two mentioned elements, the command *start* is called.

- States: Dictionary dealing with the different states previously defined. Each state is defined by the *MessageHandler.* When the state is *LOCATION* and the input match the *MessageHandler* requirement, the command *newUser* is called, while for the *IDENTIFICATION* state the command *identifyIntent* is called.
- Fallback: Defines the closing states and it is composed as the Entry-points of a *CommandHandler* accepting this time the input /*done* and a *RegexHandler* accepting inputs of type *bye.*

3.1.2. Bot functions

Here we store the auxiliary scripts that contain all necessary functions to make the Bot work. These scripts are:

- Bot Command: Function designated as command. These are the functions called by the conversation object. The specificity of these functions is that the arguments must be bot and update and nothing else.
- Bot Function: Functions which are not depending on the objects previously mentioned bot and update, such as 'buildMenu' which is not using any of the objects parameters
- Global var: File used as a setting script. It allows to change the model used to look for the answer, change the thresholds of the project and the waiting time.
- Handlers Folder: In this folder can be found the script dedicated to the different types of object need to build a dialog with the telegram python package.
    - The file speakerHelper.py is used to code all the method directly related to interact with the user.
    - The file checkerHelper.py is used to code all the method in charge of the checking.
    - The file actionsHelper.py is used to code all the method related to the action the bot should realise. For example, the welcome function is in charge of a welcome message to the user. Depending on type of user (unkown/known) variable, the message will change.
    - The file botHelper.py is used to code a 'block' object. This object is required to build a command. Indeed, each command is actually a block in which we are executing actions. These actions used some checking functions and speaker functions.

In the end, adding a new skill to the bot, is actually adding a new command. This command must be located in the bot_commands.py file. It should be a function with the argument bot and update which are the arguments required by the telegram package. Each command deals with a specific type of message and uses, as mentioned before, actions function to treat them.

3.1.3. Database folder

We can find the necessary data for the running of the Bot, including:

- Ava.sqlite. It is the SQLite DB in which the data contained will be explained in the section 3.2. of this chapter.
- Two python files. The first one (data_utils) gathers the functions related to the data preprocessing. The second (dataHandler) groups all the functions used to deal with the database.
- A jupyter notebook. To manually add data into the database

3.1.4. Models folder

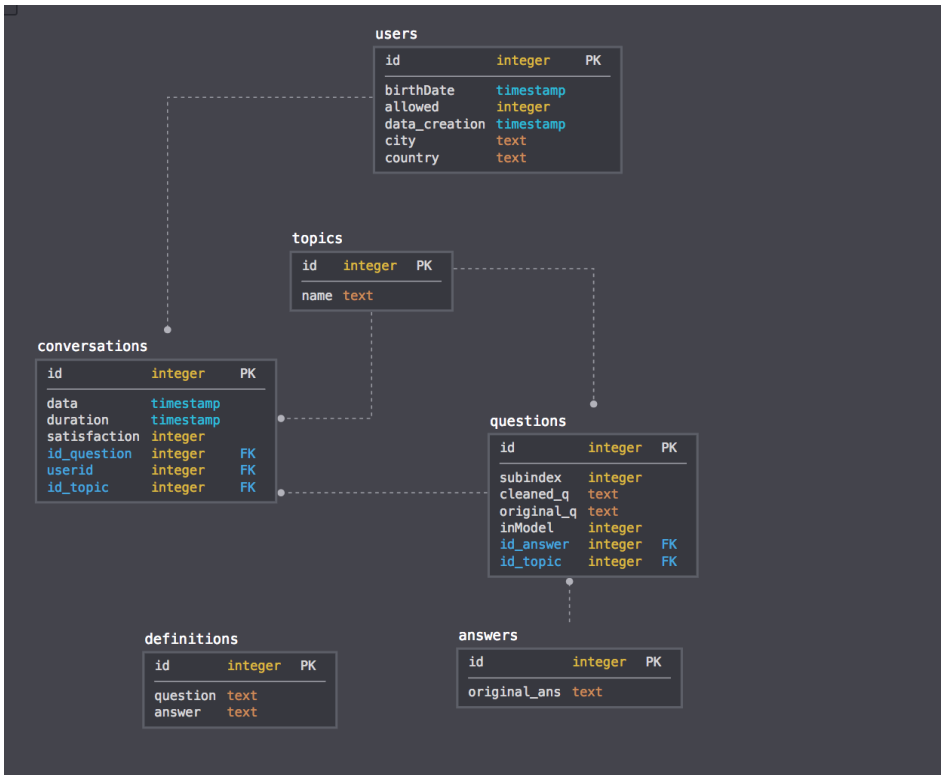
All the necessary functions and data related to the models computation can be found in this folder:

- models files. The files with the model and instances trained. They are external files which are loaded only when the Bot is started.
- Python scripts. Contain all the function to compute the models
- A jupyter notebook. To train the word2vec model.

3.2. Database

It has been decided to manage the backend with a sqlite database. Indeed, SQLite is a very light-weight relational database management system contained in a C programming library. Rather to be a client-server database engine, it is embedded into the end program.

To build the database, the python's package *sqlite3* was used. This package allows us to create and open a connection to the database from python.



The structure of the database was designed to facilitate the following points:

- ❖ the data extraction for the dashboard
- ❖ the saving of key elements of a conversation bot-user
- ❖ the training of model

Therefore, the data gather from the conversation is saving into one of the following tables:

- **User**. Containing the user information.
  - **Id**: User id, a unique id delivered by telegram for each user.
  - **Birthdata**: The birth date of the user gathers during the first connection.
  - **Allowed**: binary variable used to check if the user is allowed to talk with the bot or no.
  - **Date_creation**: The date of the user registration to the bot system.
  - **City**: User's current city.
  - **Country**: User's current country.
- **Conversations**. Record of all the conversations with the users.

    - **Id**: conversation id, which is unique.
    - **Date**: date of the conversation.
    - **Duration**: duration of the conversation, from the first greeting to the last user answer.
    - **Satisfaction**: User's satisfaction.
    - **Id_quesiton**: foreign key of the *Health question* id saved into the Questions table.
    - **Userid**: foreign key of the user id of the Users table.
    - **Id_topic**: foreign key of the topic id saved into the Topics table.

- **Questions**. Contains the questions of the dataset used in this project, but also the *Health question* gathered from the conversations with the users.
    - **Id**: unique id given to each each question.
    - **Subindex**: none unique id given for each topic. For example, four ids equal to 1 can be found into the sub-index column.
    - **Cleaned_q**: pre-processed questions as explained in the chapter 2, in the stop-words removal part.
    - **Original_q**: non pre-processed questions
    - **inModel**: Binary variable which aims to facilitate the selection of the questions to train the model.
    - **Id_answer**: foreign key of the answer's id to which is linked the question.
    - **Id_topic**: Foreign key of the topic id.

- **Topics**: Gather the information related to the dataset's topics.
    - Id: unique id for each topic.
    - Name: name of the topic.

- **Definitions**:
    - Id: unique id for each question.
    - Question: the question of type *question definition.*
    - Answer: the definition of each topic

The reason why the *question definition* type of messages was put in a separated table, is due to the way the data was created. Indeed, it was more convenient to create a table in which all the manually created were put together.

3.3. Additional Functionality

We noticed that it was important for the bot to be able to detect the language in which the user is talking. To do so, we implemented a function in the checkerHelper script, to check the words inside the input. These words will then be compared to a list of stops-words in different languages. A score is computed for each possible language and the one having the highest score is selected. If this language corresponds to English the chat flow continue, however if the language detected is not English, a message is sent to the user to remember him to speak in English.

CONCLUSION AND FUTURE WORK

In this final chapter, the conclusions are exposed, as well as the contributions made in this project and the future work proposed.

CONCLUSIONS

The initial goals of the project have been reached even though some data problems occur during the first two months and half. First, we will make a recap on the different aspects of the project done:

- **NLP approaches**: We have been able to build a system that is able to understand the meaning of the questions by using the Word Mover's Distance for the *Health questions* type and the Cosine similarity for the *questions definitions* and *already asked questions*.
- **Front-end Applications**. We have been able to implement this system into an actual text messaging application, Telegram, which is widely used, thus giving the opportunity to reach our system a huge range of users. As well as building a dashboard, using Power Bi, which shows the key elements gathered by the system.
- **Back-end Development**. We have implemented a wide range of functionalities for our Chatbot, which enriches the user experience and provides a useful service.
- **Python performance**. Now that the solution is built, we can conclude that given all what has been seen during the project, python is a very good option to develop Chatbot, both for its flexibility and the performance provided for natural language processing tasks.

Furthermore, a considerable autonomous learning has been done all along the project, which has provided lots of knowledge about many different fields and about the natural language processing world.

Taking a step back, it looks like we could have dedicated an entire Master Thesis to the natural languages approaches and another one for the Chatbot implementation. Of course, we have not been able to enter to all of these aspects at a full level of detail, but an an effort was made to build an overall solution with high performance considering the magnitude of this thesis.

FUTURE WORK

Like in all project, there are always parts that could be improved or done differently. Several axes of for improvement will be discussed here.

First concerning the NLP layer of the implemented solution, a work is currently done to improve the computation time of the Word Mover distance. This improvement is based on the paper *Linear-Complexity Relaxed Word Mover's Distance with GPU Acceleration by Kubilay Atasu, Thomas Parnell, Celestine Dünner, Manolis Sifalakis, Haralampos Pozidis, Vasileios Vasileiadis, Michail Vlachos, Cesar Berrospi, Abdel Labbi.* They transformed the Relaxed Word Mover Distance into a low-complexity implementation that reduces the average time complexity to linear. Their solution maps well onto GPUs. We tried to implement their solutions in C++ but so far, no improvement in term of computation have been realized yet.

Another idea to improve the quality of the model, is trying to find similar dataset with a similar vocabulary in order to improvement the training of the Word2vec model.

A deep learning solution, has been tried in order to generate the answers instead of retrieved. The idea is to go into a more personal conversation with the users. So far, the results didn't reach the result of the implemented model. But this could be a way to improve the bot.

From a technical point of view, it could be considered to migrate the solution to the cloud. Indeed, right now the bot is running in local and while it is a demo project, to propose a solution running into the cloud.

.

APPENDICES

For privacy issues, only code regarding the data cleaning process will be attached. Code from the Chatbot implementation can be asked for consultation for the thesis evaluation.

A. data_utils.py

This script provides the functions used to clean, analyse and correct string/words in cells of dataframe.

```
# ----------------------------------------#
# List of imports
# ----------------------------------------#


import warnings
warnings.filterwarnings(action='ignore', category=UserWarning,
module='gensim')
import re
from nltk import word_tokenize,wordpunct_tokenize
from nltk.corpus import stopwords,wordnet


# ----------------------
# Text Pre-processing
# ----------------------

def preprocess(textVariable, dictionary=None,
removeApostrophes=True, special_element=None,removeNumbers=True,
               removePunctuation=True, removeControlChars=True,
removeWhiteSpace=True,
               removeExtraSpace=True,
toLower=True,singleLetter=True,removeUnderscore=True,outSpace=False,
cleanwords=True):
    """
    This function takes as input a text variable and optionally a
dictionary. The function computes
    basic text mining pre-processing.

        :param textVariable: string used as input
        :param dictionary: only required  for cleanwords
        :param removeApostrophes: Boolean variable
        :param removeNumbers: Boolean variable
        :param removePunctuation: Boolean variable
        :param singleLetter : Boolean variable
        :param removeControlChars: Boolean variable
        :param removeWhiteSpace: Boolean variable
        :param removeExtraSpace:Boolean variable
```

```
        :param toLower: Boolean variable
        :param removeUnderscore : Boolean variable
        :param outSpace : Boolean variable
        :param cleanwords: correct misspelled words

        :return textVariable : A preprocess text variable

        """
    temp = textVariable
    # remove Apostrophes
    if removeApostrophes:
        temp = re.sub("'", "", temp)
    # remove Accent grave
    if special_element is not None:
        for elt in special_element:
            pattern = re.compile(r'\b{0}\b'.format(elt),
re.IGNORECASE)
            if re.findall(pattern, temp):
                sentence = re.sub(pattern, " ", temp)
            temp = temp.replace(elt, " ")

    # replace punctuation with space
    if removePunctuation:
        temp = re.sub(r'[^\w\s]', '', temp)
    # remove numbers
    if removeNumbers:
        temp = re.sub("\d", " ", temp)

    # remove single letter in the text
    if singleLetter:
        temp = re.sub('(\\b[A-Za-z] \\b|\\b [A-Za-z]\\b)', '', temp)

    # replace control characters with space
    if removeControlChars:
        temp = re.sub(r'[\x00-\x1f\x7f-\x9f-\xa0]', ' ', temp)

    # remove whitespace at beginning and ending of  cells
    if removeWhiteSpace:
        temp = temp.strip()

    # remove extra space in the document
    if removeExtraSpace:
        temp = re.sub("\s{2,}", " ", temp)

    # force to lowercase
    if toLower:
        temp = temp.lower()

    # remove underscore of the textVariable
    if removeUnderscore:
        temp = re.sub("_", " ", temp)


    # remove all the space of the textVariable
    if outSpace:
        temp = re.sub(" ", "_", temp)

    # correct misspelled words
    if cleanwords:
        if dictionary:
            for key in list(dictionary.keys()):
```

```
                pattern = re.compile(r'\b{0}\b'.format(key),
re.IGNORECASE)
                if re.findall(pattern, temp):
                    temp = re.sub(pattern, dictionary[key], temp)
        else:
            return print('Please add a dictionary to correct
misspelled words or mark cleanwords=False')

    return temp

# ----------------------------------------#
# General Cleaning
# ----------------------------------------#
def calculate_languages_ratios(sentence,language_used=None,stop_list=None):
    """
        Calculate probability of given text to be written in several
languages and
        return a dictionary that looks like {'french': 2, 'spanish':
4, 'english': 0}

        :param sentence: Text whose language want to be detected
        :type text: str

        :return: Dictionary with languages and unique stopwords seen
in analyzed text
        :rtype: dict

        nltk.wordpunct_tokenize() splits all punctuations into
separate tokens
        """
    languages_ratios = {}
    tokens = wordpunct_tokenize(sentence)
    words = [word.lower() for word in tokens]

    # Compute per language included in nltk number of unique
stopwords appearing in analyzed text
    for language in stopwords.fileids():
        if language_used is not None and stop_list is not None:
            stopwords_set = set(stop_list +
stopwords.words(language))
        else:
            stopwords_set = set(stopwords.words(language))

        words_set = set(words)
        common_elements = words_set.intersection(stopwords_set)
        languages_ratios[language] = len(common_elements)

    return languages_ratios

def detect_language(text, language=None,stop_list=None):
    """
    Calculate probability of given text to be written in several
languages and
    return the highest scored.

    It uses a stopwords based approach, counting how many unique
stopwords
    are seen in analyzed text.

    :param text: Text whose language want to be detected
    :type text: str
```

```
    :return: Most scored language guessed
    :rtype: str
    """

    #list of language to consider as english text
    as_english = ['azerbaijani','danish', 'dutch']

    ratios =
calculate_languages_ratios(text,language_used=language,stop_list=sto
p_list)
    most_rated_language=max(ratios, key=ratios.get)

    if len(set(ratios.values()))==1 or most_rated_language in
as_english:
        most_rated_language=language

    return most_rated_language,ratios



defGeneralCleaning(index,listText,check_reject,language=None,stop_list=Non
e):
    # remove repeated character (which are not alphanumeric)
    listText = [re.sub('([^\w\s]{2,})', '', str(sentence)) for
sentence in listText]

    # remove repeated character in list [a-z] and replace it with
just one ex aaaaaaaah --> ah
    listText = [re.sub('((?![o])[a-z])\1{1,}',
'\1',str(sentence))for sentence in listText]

    # remove date and time
    listText = [re.sub('[0-9]{2}[\/,:.][0-9]{2}[\/,:.][0-9]{2,4}',
'', str(sentence))
        for sentence in listText]


    # remove special character  <>#{}€%~€™
    listText = [re.sub('[<>#{}€%~€™Ã©Â¨Ãª]', '', str(sentence))
                for sentence in listText]

    # replace \b(Ã©|Ãª|Ã¨)\b by e
    listText = [re.sub('\b(Ã©|Ãª|Ã¨)\b', 'e', str(sentence))
            for sentence in listText]

    # remove special character ]{>()}<[*?
    listText = [re.sub('[]{>()}<[*?]', '', str(sentence))
                for sentence in listText]

    listText = [preprocess(str(sentence), removeApostrophes=False,
                                removeNumbers=False,
removePunctuation=False,
                                removeControlChars=True,
removeWhiteSpace=True,
                                removeExtraSpace=True,
toLower=False, singleLetter=False,
                                removeUnderscore=True,
outSpace=False, cleanwords=False)
                        for sentence in listText]
```

```
    if language is not None:
        if check_reject is None:
            check_reject={}
        listText_checking=[]
        # detect sentence with none english language
        for i,sentence in enumerate(listText):
            language_detected,ratios=detect_language(sentence,
language=language,stop_list=stop_list)
            if language_detected == language :
                listText_checking.append(sentence)
            else:
                print("#------------------------#")
                print(language_detected)
                print("#------------------------#")
                print()
                key=check_reject.setdefault(i,None)
                if key is not None:
                    check_reject[i].append(language_detected)
                else:
                    check_reject[i]=[language_detected]

        # return empty list if one of the sentence written in
language
        if len(listText)!= len(listText_checking):
            for i in range(len(listText)):
                listText[i]=""

    return listText,check_reject


def misspelled_words(tokens):
    '''
    function to correct the spelling of words(tokens)
    :param tokens: list of token to check
    :return: tokens_correct list of token with correct spelling
    '''
    import re
    from autocorrect import spell
    tokens_correct = []
    for token in tokens:
        if not re.findall(re.compile(r'[^\w\s]'), token):
            tokens_correct.append(spell(token))
        else:
            if len(tokens_correct) > 1:

tokens_correct.append(''.join([tokens_correct.pop(tokens_correct.ind
ex(tokens_correct[-1])), token]))

    # tokens_correct=[spell(token)for token in tokens if not
re.findall(re.compile(r'[^\w\s]'),token) ]
    return tokens_correct


def removeElement(sentence, replaceObj, dict_arg=None):
    import re
    if isinstance(replaceObj, dict):
        # case 1 : replaceDict={'by':'toReplace'}
        # -->replace one word/expression('toReplace') by a specific
word/expression ('by')
        # case 2 : replaceDict={by:'[toReplace1,toReplace2]'}
        # -->replace list of
```

```
words/expressions8[toReplace1,toReplace2]) by a specific
word/expression ('by')
        # toReplace can be a regex expression like [\(\)\{\}<>]
        for by in replaceObj.keys():
            toReplace = replaceObj[by]
            if isinstance(toReplace, list):  # case 2
                pattern = re.compile("|".join(toReplace),
re.IGNORECASE)  # "({})".format(x) for x in toReplace)
                sentence = re.sub(pattern, by, sentence)
            else:
                sentence = re.sub(re.compile(toReplace,
re.IGNORECASE), by, sentence)

    if callable(replaceObj):
        def replace_with_function(sentence,replaceObj,*args):
            if len(args) < 2:
                return print('Please give me a regex expression and
a word to replace')
            if replaceObj.__name__ == 'search':
                if len(args) == 3:  # args =['regex
expression','by',index]
                    elt = re.search(args[0], sentence,re.M |
re.IGNORECASE)
                    if elt is not None:
                        elt=elt.group(args[2])
                else:
                    return print(
                        "-->Please add a boolean value (0 or 1) for
the group function. Ex: removeElement(tt,re.search,r'[0-
9]','feet',1)")
            else:
                elt = replaceObj(args[0], sentence,re.M |
re.IGNORECASE)

            if isinstance(elt, list) and elt is not None:
                pattern = re.compile("|".join(elt), re.IGNORECASE)

            else:
                if elt is None or elt=='':
                    pattern = ''
                else:
                    pattern = re.compile(elt, re.IGNORECASE)

            sentence = re.sub(pattern, args[1], sentence)
            return sentence

        def reformatList(key, listValues):
            if len(listValues) > 1:
                index = listValues[1]
                elt_list = [listValues[0], key, index]
            else:
                elt_list = [listValues[0], key]
            return elt_list

        if dict_arg is not None:
            for by in dict_arg.keys():
                values = dict_arg[by]
                if isinstance(values,list): # dict type =
{'key':[values],[values],..]
                    for i in values:
                        if isinstance(i,list):# dict type =
```

```
{'key':[[value1,value2],[value1,value2],..]
                            elt_list=reformatList(by, i)
                            sentence =
replace_with_function(sentence, replaceObj, *elt_list)
                        else:
                            elt_list=[i,by]
                            sentence =
replace_with_function(sentence, replaceObj, *elt_list)
                else:
                    elt_list = [values, by]
                    sentence =
replace_with_function(sentence,replaceObj, *elt_list)
    return sentence


# ----------------------------------------#
# Replacing Words Matching Regular Expressions
# ----------------------------------------#

replacement_patterns = [
    (r'won\'t', 'will not'),
    (r'can\'t', 'cannot'),
    (r'i\'m', 'i am'),
    (r'ain\'t', 'is not'),
    (r'(\w+)\'ll', '\g<1> will'),
    (r'(\w+)n\'t', '\g<1> not'),
    (r'(\w+)\'ve', '\g<1> have'),
    (r'(\w+)\'s', '\g<1> is'),
    (r'(\w+)\'re', '\g<1> are'),
    (r'(\w+)\'d', '\g<1> would'),
]

class RegexpReplacer(object):
    """ Replaces regular expression in a text.
    replacer = RegexpReplacer()
    replacer.replace("can't is a contraction")
    'cannot is a contraction'
    replacer.replace("I should've done that thing I didn't do")
    'I should have done that thing I did not do'
    """

    def __init__(self, patterns=replacement_patterns):
        self.patterns = [(re.compile(regex), repl) for (regex, repl)
in patterns]

    def replace(self, text):
        s = text

        for (pattern, repl) in self.patterns:
            s = re.sub(pattern, repl, s)

        return s


class RepeatReplacer(object):
    """ Removes repeating characters until a valid word is found.
    >>> replacer = RepeatReplacer()
    >>> replacer.replace('looooove')
    'love'
    >>> replacer.replace('oooooh')
    'ooh'
```

```
    >>> replacer.replace('goose')
    'goose'
    """

    def __init__(self):
        self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)')
        self.repl = r'\1\2\3'

    def replace(self, word):
        if wordnet.synsets(word):
            return word

        repl_word = self.repeat_regexp.sub(self.repl, word)

        if repl_word != word:
            return self.replace(repl_word)
        else:
            return repl_word


def preprocess_doc(documents, language='english'):
    '''
    Function to clean the messages.
    :param documents: message (string) to clean
    :param language: language to use for stop_word
    :return: cleand tokenized message
    '''
    # Lower the text.
    doc = documents.lower()

    replacer = RegexpReplacer()
    replacer.replace(str(doc))

    # Remove stopwords.
    stop_words = stopwords.words(language)
    doc = word_tokenize(doc)  # Split into words.
    doc = [w for w in doc if not w in stop_words]  # Remove
stopwords.
    doc = [w for w in doc if w.isalpha()]  # Remove numbers and
punctuation.

    return doc
```

B. Dataset creation

In this part, the python steps to create the extract the data from the json files and create a csv file are presented :

```
# #### Import packages

import os, json
import pandas as pd


# #### Load files and create dataset
path_to_json = './json/'

# Load files' names into a list
json_files = [pos_json for pos_json in os.listdir(path_to_json) if
pos_json.endswith('.json')]
print("Files' names: ",json_files)


# Create an empty dataset
jsons_data = pd.DataFrame(columns=['question', 'answer', 'tags'])

## Fill dataset

#initialised indexes of dataset
index = 0

# loop over names' files
for js in json_files:
    with open(os.path.join(path_to_json, js)) as json_file:
        json_text = json.load(json_file)

        #Look for considered elements
        for elt in json_text:
            question = elt['question']
            answer = elt['answer']
            tags = elt['tags']

            # Push a list of data into the DataFrame at row given by
'index'
            jsons_data.loc[index] = [question, answer, tags]
            index+=1


#save dataset
jsons_data.to_csv('dataset_QnA.csv', sep=';', encoding='utf-8',
index=False)
```

C. Data cleaning

This part exposed the functions and steps realised to clean the data, corresponding to the part 2.1. of the second chapter.

a. Functions used in the cleaning process

In this part, funcitons used for the cleaning realised on the extracted data.

```
# ----------------------------------------#
# List of imports
# ----------------------------------------#
#package
import pandas as pd
#import bot_data.data_utils
import data_utils


# ----------------------------------------#
# Functions
# ----------------------------------------#

def removeRows(dataframe,columns):
    '''
    function to remove rows where one of the variables contained in
columns list is empty
    :param dataframe: pandas dataframe
    :param columns: columns to consider
    GoodDataFrame and BadDataFrame a
    :return: -
    '''
    #initialize list toRemove
    index_list_toRemove=[]

    # cell with len <=1 replace by ""
    for index, row in dataframe.iterrows():
        add = False
        toCheck = row[columns].tolist()
        for i,sentence in enumerate(toCheck):
            if (len(str(sentence))<=1) or sentence=="N/a" or
sentence==' ' :
                dataframe.loc[index, columns[i]] = ""
                add = True
            if pd.isnull(sentence):
                dataframe.loc[index, columns[i]] = ""
                add = True
            if add and  index not in index_list_toRemove:
                index_list_toRemove.append(index)

    #remove element contained in index_list_toRemove
    dataframe.drop(dataframe.index[index_list_toRemove],
inplace=True)
    print('rows removed')
    return dataframe
```

```
def removeDuplicateRow(dataframe,path, *args,merge=True):
    '''
    function to remove duplicate row having same element in column A
and B
    :param dataframe: pandas dataframe
    :param columns: columns to consider
    :param args: list of two columns to be considered

    save dataframe without duplicate rows
    :return: -
    '''

    df = pd.read_csv('{0}/'.format(path) + dataframe, sep=";")

    columnA = args[0]
    columnB = args[1]

    # first drop rows with Q and A equal (we keep the last)
    newdf = df.drop_duplicates(subset=[columnA, columnB],
keep='last')

    # merge question duplicate with different answer
    list_trueFalse= newdf.duplicated([columnA], keep=False).tolist()
    index_duplicate_row=[index for index, elt in
enumerate(list_trueFalse) if elt]

    dict_treat_row = {}
    index_list_toRemove = []
    for index in index_duplicate_row:
        question = newdf.loc[newdf.index[index], columnA]
        answer = newdf.loc[newdf.index[index], columnB]

        if question in dict_treat_row.keys():
            dict_treat_row[question].append(answer)
            index_list_toRemove.append(index)
        else:
            dict_treat_row[question] = [answer]

    # remove element contained in index_list_toRemove
    newdf = newdf.drop(newdf.index[index_list_toRemove])

    #fill the new dataset
    for index,row in newdf.iterrows():
        question = row[columnA]
        if question in dict_treat_row.keys():
            if merge:
                row[columnB] = '/'.join(str(v) for v in
dict_treat_row[question]) #merge all answer into a string
            else:
                row[columnB]= dict_treat_row[question][-1] #take the
last element

    newdf.to_csv('{0}/3_removeDuplicateRow.csv'.format(path),
sep=';', encoding='utf-8', index=False)
    print('You removed {0} rows'.format(df.shape[0] -
newdf.shape[0]))
```

```
def spellingCorrection(dataframe, columns, path, name=None):
    '''
    :param dataframe: pandas dataframe
    :param columns: columns to check
    :param path: where to save dataframe
    :param name: if name not none, save the dataframe in path
    :return: dataframe with correct spelling in columns
    '''
    from  data_utils import misspelled_words
    import nltk
    dataframe.reset_index()
    for index, row in dataframe.iterrows():
        to_check = row[columns].tolist()
        for column, sentence in enumerate(to_check):
            tokens = nltk.tokenize.word_tokenize(sentence)
            tokens_correct = misspelled_words(tokens)
            dataframe.at[index, columns[column]] = ' '.join(str(v)
for v in tokens_correct)
        print('Row : %s corrected' % str(index), 'Shape dataframe',
dataframe.shape[0])

    dataframe = removeRows(dataframe, columns)

    print('-----------------------------------')
    print('All the columns have been corrected')
    print('-----------------------------------')

    if name is not None:
        dataframe.to_csv('{0}'.format(path) + name + '.csv',
sep=';', encoding='utf-8', index=False)
        print('Dataframe saved')

    return dataframe

def NameDictionary(dataframe,columns,path,name_dict=None):
    '''
    Function to identify the name in a sentence
    :param dataframe: pandas dataframe
    :param columns: columns list to check
    :param path: where to save the dictionary
    :param name_dict: name of dictionary to save
    :return: dictionary {'Jo':[(23,'Q')]} = {'Name':[(index in
dataframe,column)]}
    '''
    import nltk
    import numpy as np
    from nltk.tag.stanford import StanfordNERTagger

    # Change the path according to your system
    stanford_classifier = 'C:/Users/charline.mas/stanford-ner-2018-
02-27/classifiers/english.all.3class.distsim.crf.ser.gz'
    stanford_ner_path = 'C:/Users/charline.mas/stanford-ner-2018-02-
27/stanford-ner.jar'

    # Creating Tagger Object
    st = StanfordNERTagger(stanford_classifier, stanford_ner_path,
encoding='utf-8')

    #creation dicitonary of the found names
    dict_name = {}
    dataframe.reset_index()
```

```
    for index,row in dataframe.iterrows():
        to_check= row[columns].tolist()
        for column,sentence in enumerate(to_check):
            for sent in nltk.sent_tokenize(sentence):
                tokens = nltk.tokenize.word_tokenize(sent)
                tags = st.tag(tokens)
                for tag in tags:
                    if tag[1] == 'PERSON':
                        print(tag,index,columns[column])
                        name=tag[0]
                        if name in dict_name.keys():
                            dict_name_temp={}
                            old_val=dict_name[name]
                            value_toAdd=[(index,columns[column])]
                            dict_name_temp[name]=old_val+value_toAdd
                            dict_name.update(dict_name_temp)
                        else:

dict_name[name]=[(index,columns[column])]

    np.save('{0}{1}.npy'.format(path,name_dict), dict_name)
    print('-----------------------------------')
    print('Dictionary names saved')
    print('-----------------------------------')
    return dict_name


def deleteElement
(dataframe,columns,replaceObj,dict_args=None,name=None,removeSentenc
e=False):

    import nltk
    from data_utils import removeElement
    dataframe.reset_index()
    for index, row in dataframe.iterrows():
        to_check = row[columns].tolist()
        for column, sentence in enumerate(to_check):

            if removeSentence:
                new_text=[]
                for sent in nltk.sent_tokenize(sentence):
                    if dict_args is not None:
                        s =
removeElement(sent,replaceObj,dict_arg=dict_args)
                    else:
                        s = removeElement(sent, replaceObj)
                    if not len(s)<len(sent):
                        new_text.append(sent.capitalize())

                sentence= ' '.join(new_text)
            else:
                if dict_args is not None:
                    sentence = removeElement(sentence, replaceObj,
dict_arg=dict_args)
                else:
                    sentence = removeElement(sentence, replaceObj)

            dataframe.at[index, columns[column]] = sentence

    if removeSentence:
```

```
            print('-----------------------------------')
            print('Sentence(s) removed')
            print('-----------------------------------')
        else:
            print('-----------------------------------')
            print('Element(s) removed')
            print('-----------------------------------')


        if name is not None:
            dataframe.to_csv('./data/'+name + '.csv', sep=';',
encoding='utf-8', index=False)
            print('Dataframe saved')

        return dataframe
```

b. Cleaning steps

```
# Import packages

import pandas as pd
import re
import nltk
from nltk  import word_tokenize
from collections import Counter
import collections
from data_extraction import *
from data_utils import *



# Load Data
df = pd.read_csv('dataset_QnA.csv', sep=";")


# General cleaning

# Remove all the duplicate and empty rows
path = './'
removeDuplicateRow('dataset_QnA.csv',path, 'question',
'answer',merge=False)

# import the previous cleaning dataset
df1 = pd.read_csv('{0}3_removeDuplicateRow.csv'.format(path),
sep=";")


# NLP cleaning
'''
Clean sentences from misspelled words, control characters, whitespace
at beginning and ending of sentence,
extra space and underscore
'''

# Remove the sentence in answer variable containing the following
elements
elt_forSentenceRemoving=['regards', 'thanks', "pictures","thankyou",
"thnkayou", 'thank you','Thnkayou','sincerely','take care']
df2 = deleteElement
(df1,['answer'],{'':elt_forSentenceRemoving},name=None,removeSentenc
e=True)


# Replace elements in the question and answer columns
courtesy = ["good afternoon", "good morning", "best regards",
"hello",
            "hi", "good evening","as above","Sorry","please"]
courtesy = [r'\bw\b' for w in courtesy]

dd={'and': '&', 'emergency room':r'\ber\b', 'you':r'\bu\b',

'which':r'\bwch\b','radiotherapy':r'\brt\b','toxicity':r'\btx\b','th
anks':r'\btxs\b','I have':r'\bive\b',
    'doctor':'dr.','feet':'ft','primary care
physician':r'\bpcp\b','electrocardiogram':[r'\bekg\b',r'\becg\b',r'\
bekgs\b'],
```

```
    'problems':r'\bpblms\b','problem':r'\bpblm\b','weight
management':r'\bwt
mngmt\b','years':[r'\byrs\b',r'\byr\b'],'i.e.':r'\b i. e \b',
    'hour':r'\bhr\b',"I don't know":r'\bidk\b','wide local
excision':r'\bwle\b','please':r'\bplz\b','abut':r'\babout\b',
    '':courtesy+[r'[\(\)\{\}<>]',"'",'-'],' ':'/'}

df3 = deleteElement
(df2,['question','answer'],dd,removeSentence=False)


# Columns to consider in the cleaning
columns=['question','answer']

# Create a copy of the previous cleaned dataset
df4 = df3.copy()
df4.reset_index()

# Apply the preprocess function to each column and each row of the
selected column.
for column in columns:
    # Select the column and defined it as a list
    S = df4[column].tolist()
    # Apply functions
    new_s = [preprocess(sentence,
removeApostrophes=False,special_element=["``",'"'],
removeNumbers=False,
                        removePunctuation=False, removeControlChars=True,
removeWhiteSpace=True,
                        removeExtraSpace=True,
toLower=False,singleLetter=False,removeUnderscore=True,
                             outSpace=False,cleanwords=False) for
sentence in S]
    new_s=[' '.join([s.capitalize() for s in
nltk.sent_tokenize(sentence)])
           for sentence in new_s]
    # Finally we add back the cleaned column to the dataset
    df4[column]=new_s


'''
Remove duplicate words ex you mention is is rated --> you mention is
rated
'''
# Columns to consider in the cleaning
columns=['question','answer']

# Create a copy of the previous cleaned dataset
df5 = df4.copy()
df5.reset_index()

for column in columns:
    S = df5[column].tolist()
    new_sentence=[]
    for index,sentence in enumerate(S):
        new_sent=[]
        for sent in nltk.sent_tokenize(sentence):
            words = sent.split(' ')
            words_copy = words.copy()
            for i in range(len(words)):
                if i != len(words)-1 and words[i]==words[i+1]:
```

```
                    words_copy.remove(words[i])
            new_sent.append(' '.join(words_copy))
        new_sentence.append(' '.join(new_sent))
    df5[column]=new_sentence


'''
Remove the whitespace created by the previous cleaning
'''
# Columns to consider in the cleaning
columns=['question','answer']

# Create a copy of the previous cleaned dataset
df6 = df5.copy()
df6.reset_index()

for column in columns:
    S = df6[column].tolist()
    new_s = [preprocess(sentence, dictionary=None,
removeApostrophes=False,special_element=None, removeNumbers=False,
                    removePunctuation=False, removeControlChars=True,
removeWhiteSpace=True,
                    removeExtraSpace=True,
toLower=False,singleLetter=False,removeUnderscore=True,
                         outSpace=False, cleanwords=False) for
sentence in S]
    new_s=[' '.join([s.capitalize() for s in
nltk.sent_tokenize(sentence)])
           for sentence in new_s]
    df6[column]=new_s



# Remove the empty rows created by the previous cleaning
df7 = removeRows(df6, columns)

# delete the following elements in the tags columns
df8 = deleteElement
(df7,['tags'],{'':[r'[\(\)\{\}\[\]]',"'"]},removeSentence=False)


'''
Remove the whitespace created by the previous cleaning
'''

# Create a copy of the previous cleaned dataset
df9 = df8.copy()
df9.reset_index()

S = df9['tags'].tolist()
new_s=[[word for word in elt.split(', ')] for elt in S]
new_s=[' '.join([s.capitalize() for s in sentence])
           for sentence in new_s]
df9['tags']=new_s

'''
Final checking to find empty rows
'''
# Columns to consider in the cleaning
columns=['tags']
```

```
# Create a copy of the previous cleaned dataset
df10 = df9.copy()
df10.reset_index()

# Initialisation of the index list of the element to remove
index_list_toRemove=[]
# Iterate through the dataset to find index of empty rows
for index, row in df10.iterrows():
    add = False
    toCheck = row[columns].tolist()
    toCheck = toCheck[0].split(', ')
    if not toCheck or ' '.join(toCheck)==' ':
        index_list_toRemove.append(index)

#remove element contained in index_list_toRemove
df10.drop(df10.index[index_list_toRemove], inplace=True)
```

c. Data selection

In this section, it is showed how we selected only the rows having one topic

```
# Transform tags columns into list
tags = df10['tags'].tolist()
new_tags = [[word for word in tag.split(', ') ] for tag in tags]

# Compute lenght of each tag_list
tag_list_lenght = list(map(len,new_tags))

# Dictionary of topics frequency
count = Counter(tag_list_lenght)

# Sorted the previously found dictionary
od = collections.OrderedDict(sorted(count.items()))


# Create index dictionary key = number of tag, values = list of
index
dd_index={}
for i in count.keys():
    dd_index[i]=[n for n, x in enumerate(tag_list_lenght) if x == i]


# Select the rows with just one tag
dataset_one_tag = df1.loc[dd_index[1],:]

#List of retained tag
tags_retained=['bariatrics','cardiac
electrophysiology','cardiology','breast surgery']

# Select rows which contained only the tags in tags_retained
dataset_retained_tags=dataset_one_tag.loc[dataset_one_tag['tags'].is
in(tags_retained)]


# Transform topic name into cluster number
df_final = dataset_retained_tags.copy()

#Create dictionary with the topics code
dict_cluster={'bariatrics':0,'breast surgery':1,'cardiac
electrophysiology':2,'cardiology':3}

# Change the tag name by the cluster number
df_final['tags'] = df_final['tags'].map(dict_cluster)


# Save dataset
df_final.to_csv('labeled_data.csv', sep=';', encoding='utf-8',
index=False)
```

BIBLIOGRAPHY

[1] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[2] *M*. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger. From word embeddings to document distances. In Proceedings of the 32nd International Conference on Machine Learning (ICML 2015), pages 957–966, 2015

[3] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In ICML, volume 14, pages 1188–1196, 2014.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representa- tions in vector space. arXiv preprint arXiv:1301.3781, 2013.

[5] C. Musto, G. Semeraro, M. de Gemmis, and P. Lops. Learning word embeddings from wikipedia for content-based recommender systems. In European Conference on Informa- tion Retrieval, pages 729–734. Springer, 2016.

[6] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. International journal of computer vision, 40(2):99–121, 2000.

[7] Tomoyosi Akiba, Katunobu Itou, and Atsushi Fujii. "Question Answering Us- ing" Common Sense" and Utility Maximization Principle." In: NTCIR. 2004.

[8] Amara D. Angelica. "How Watson Works: a conversation with Eric Brown, IBM Research Manager". In: Kurzweil Accelerating Intelligence (2011). URL: http://www.kurzweilai.net/how-watson-works-a- conversation-with-eric-brown-ibm-research-manager.

[9] Ricardo Baeza-Yates and William Bruce Frakes. Information retrieval: data structures & algorithms. Prentice Hall, 1992.

[10] Yoshua Bengio et al. "A neural probabilistic language model". In: Journal of machine learning research 3.Feb (2003), pp. 1137–1155.

[11] Mark Clark. A Chatbot Framework. 2016. URL: http://info.contactsolutions.com/digital-engagement-blog/a-chatbot-framework.

[12] Ralf Herbrich and Thore Graepel, Handbook of natural language processing second edition. Microsoft research, Cambridge, UK. Part I: Classical approaches.

[13] Document-term matrix. (2015, December 23). Document-term matrix — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Document-term_matrix

[14] Card, S. K., Robertson, G. G., and Mackinlay, J. D. (1991). The information visualizer: An information workspace. Proc. ACM CHI'91 Conf. (New Orleans, LA, 28 April-2 May), 181-188.

[15] Miller, R. B. (1968). Response time in man-computer conversational transactions. Proc. AFIPS Fall Joint Computer Conference Vol. 33, 267-277.

[16] Myers, B. A. (1985). The importance of percent-done progress indicators for computer-human interfaces. Proc. ACM CHI'85 Conf. (San Francisco, CA, 14-18 April), 11-17.

[17] Miteva, S. (2017, August 21). 5 Programming Languages You Can Use to Create Chatbots. http://valosohub.com/blog/2017/08/21/programming-languages-chatbots/

[18] Willems, K. (2015, May 12). Choosing R or Python for Data Analysis? An Infographic. https://www.datacamp.com/community/tutorials/r-or-python-for-data-analysis

[19] Kopf, D. (2017, September 22). If you want to upgrade your data analysis skills, which programming language should you learn ?. https://qz.com/1063071/the-great-r-versus-python-for-data-science-debate/

[20] Lobo, J. (2017, August 16). How to choose the best channel for your chatbot. https://www.inbenta.com/en/blog/chatbot-choose-best-channel-chatbot/

[21] Briz, V. (2017, March 29). The 3 Best Platforms for your ChatBot. https://chatbotslife.com/the-3-best-platforms-for-your-chatbot-d2693289950d

[22] Bearon, J. (2016, December 2). How to choose the best channel for your chatbot. https://recast.ai/blog/how-to-choose-the-best-channel-for-your-bot-the- ultimate-cheat-sheet/

[23] Brisson, K. (2016, August 25). 11 best messaging platforms for your chatbot or conversational app. https://blog.init.ai/pick-your-platform-wisely-c5ab5bc7555d

[24] Dogtiev, A. (2017, December 5). Telegram Revenue and Usage Statistics. http://www.businessofapps.com/data/telegram-statistics/

[25] Telegram (messaging service). (2018, January 1). Telegram (messaging service) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Telegram_(messaging_service)

[26] Karush–Kuhn–Tucker conditions. (2017, December 1). Karush–Kuhn–Tucker conditions — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Karush-Kuhn-Tucker_conditions

[27] Dwyer, G. (2016, November 10). Building a Chatbot using Telegram and Python (Part 1). https://www.codementor.io/garethdwyer/building-a-telegram-bot-using-python- part-1-goi5fncay

[28] Push technology. (2018, January 2). Push technology — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Push_technology#Long_polling