SAPIENZA
UNIVERSITÀ DI ROMA

Text Extraction with POS-Tagging and Word Embeddings for support tickets automation

Faculty of Information Engineering, Computer Science and Statistics

Corso di Laurea Magistrale in Data Science

Candidate

Pol Ribó

ID number 1840853

Thesis Advisor

Prof. Simone Scardapane

Co-Advisor

Alberto Massidda

Academic Year 2019/2020

Thesis defended on 27 May 2021
in front of a Board of Examiners composed by:

Prof. Anagnostopoulos Aris (chairman)
Prof. Brutti Pierpaolo
Prof. Galasso Fabio
Prof. Maggino Filomena
Prof. Palagi Laura
Prof. Petti Manuela
Prof. Scardapane Simone

Text Extraction with POS-Tagging and Word Embeddings for support tickets automation

Master's thesis. Sapienza – University of Rome

© 2021 Pol Ribó. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: riboleon.1840853@studenti.uniroma1.it

Abstract

The ability of automating processes is usually high-valued in businesses, as the benefits that it provides are many, such as productivity increases, saving costs or reducing working hours. However, in the context of technological and, more precisely, software companies, the nature of the tasks or processes suitable to be automated is vast, since it depends on factors that at its due time depend on environmental conditions; that would be for example the specific area of interest of the company, its target market or the company mission and vision. Despite that fact, Machine Learning and Deep Learning powered models and systems can be used as tools to make automation happen faster and more effective. Applying the correct models and more importantly making these models right is crucial for any attempt of a successful automating task.

Furthermore, the goals of this thesis are, on general terms:

(i) An evaluation of how to approach to automating a keyword extraction task, its caveats and restrictions and the motivation behind.

(ii) Research and selection of a Machine Learning powered approach to deal with a concrete business process automating problem involving keyword extraction. Since the approach involves the implementation of several Natural Language Processing techniques, a complete analysis of them is also performed.

(iii) Explanation of the implementation and evaluation of the ensemble model in form of a pipeline specifically built for this task in order to extract keywords.

(iv) Conclusions after the completion of the task, available room for improvement and estimation of the potential of keyword extraction.

So, if we want to narrow down the above bullet points, we inevitably need to talk about the nature of the task and also the research done. The solutions for which to solve automation are many and can come in various forms. The proper formulation of that is critical for the project, and all the model and its implementation turns around its proper understanding, as it is perceived across this very document. Despite that, the use for Machine Learning and Deep Learning is unquestionable given the cause of this project and as a consequence of the character of the process to be automated, which involves support tickets and therefore Natural Language Processing. Hence, a thorough description of such techniques has been carried out and how they are linked to the end goal of the project.

As far as the practical part of the project, there are three techniques that need to be mentioned; that would be Part-Of-Speech Tagging (POS-Tagging), Dependency Parsing and Word Embeddings (WE). All are absolutely fundamental for the project and its performance is key for a correct functioning of the keyword extraction pipeline, therefore an analysis of the models as well as the interpretability of them were performed, and these includes plots, confusion matrix, etc.. it is worth mention that before the decision to implement them was made, other NLP techniques were researched such as Named Entity Recognition or the construction of Knowledge Graphs.

Regarding the final model, there is a complete explanation of how the NLP techniques are ensembled into a pipeline and some results that have been executed on artificially built tickets.

The conclusions of the project take into consideration possible improvements that could be done on the NLP field, and since the main purpose of the project is to extract words from a random customer support ticket, how the pipeline could be scaled up to retrieve any given word and adressing future challenges.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Gordon Moore, co-founder of Intel, is also known for giving birth to Moore's Law [1]. Moore's Law states the visionary perception of Gordon that implied the number of transistors on a microchip would double every two years, though the cost of computers would be halved which, plainly speaking, means that we will have more powerful computers at less cost. How can this be possible, what made one of the fathers of Intel capable of coming up with this theory? Of course, the answer is the belief that technology would exponentially improve. And time has proven him right, so it can be derived that investing in developing new technologies that are able to substitute or improve hand-work is desirable, and this precise standpoint justifies the main purpose of this document and the work behind it, which is non other than the improvement of a technological service in order to reduce costs.

Businesses are always keen to learn new ways with which to increase the value of its products or services. If we speak of technological companies, the need to continually reinvent themselves so as to respond and adapt to the needs of an endlessly innovating market, that thrive becomes more apparent. And, if we make the quest even tighter and refer to software companies, it is a matter also of growth. This is the mindset that the people of Sourcesense have and the reason why of this project, which was done in form of an internship in the company. Sourcesense is a software technology company with more than 15 years of experience, based in Italy but has offices also in the UK, that describes itself as a [2] "company with the aim of promoting the use of Open Source software in the Enterprise area". With this in mind, a common way to create value is process automation.

Business process automation, or Process Automation is the technology-enabled automation of complex business processes. Its a wide concept, since it can streamline a business for simplicity, achieve digital transformation, increase service quality, improve service delivery or contain costs. Plainly, it is the use of technology to execute recurring tasks or processes in a business where some amount of manual effort can be replaced. It consists of integrating applications, restructuring labor resources and using software applications throughout the organization. So, all in all, from the business efficiency viewpoint, automating processes is a good idea most of the times. Benefits can be of different nature, such as productivity increases, saving costs, reducing working hours...

Nevertheless, if we speak in the concrete case that this document is describing, the process that needs to be automated is related to customer support tickets. The term "support ticket" describes the interaction between a customer and a service representative. It's the basic element of any customer experience related job—allowing a business to create, update, and hopefully resolve any issues the end-users might have. In this document, the automation of support tickets will be directly translated to less working hours by achieving the parsing of the tickets and also filling missing information by replying to the customer in the very same moment; otherwise that could take hours or even days if we take into account that the customer may reply only once a day. Technically speaking, the time for the step of information procurement for later use is dramatically diminished. In addition, simple tasks can be executed out of information input in a form ; once the information is procured in the first step, the company could just automatically forward it in that mentioned form and have a program perform any actions needed, which in turn would also save many working hours and increase efficiency.

Once the main headlines have been described, the path towards the completion of this work can be summarized in some nuclear points. On the starting line, the talking points were around both the specifications of the task to be automated and getting at the big picture so that the focus of the research becomes as narrow as it can be. However, in the end several points of view needed to be taken into consideration which led to the reading and comprehension of general and concrete frameworks, single and combinative approaches, old and novel techniques and above all the continuing questioning about its feasibility. It is important to state that although now feasibility may not sound that important or relevant, it proved to be key because it conditions all the future steps. At the end, two main techniques were found the most suitable: we talk about Part-of-Speech-Tagging (POS-Tagging) and

Word Embeddings(WE), both found in the field of Natural Language Processing (NLP).

Hence, Natural Language Processing, or NLP for short, is broadly defined as the automatic manipulation of natural language, like speech and text, by software. The growth of this field has been evolving constantly throughout history but reached a promising turning point with the rising of Deep Learning, allowing the creation of much more sophisticated applications. For example, great results have been achieved on various NLP tasks such as visual question answering (QA),machine translation or the aforementioned Part-of-Speech Tagging and Word Embeddings.

POS-Tagging is the task of assigning to each word in a corpus a tag regarding its lexical nature. Given such, they can give information about a word and its neighbors. For example if a tag refers to "adverb", it can be inferred that there is a verb in a sentence. POS-tags are used in corpus searches and in text analysis tools and algorithms, as is the case of this work. Regarding to WE, it is a technique that builds distributional vectors for a word, based on the principle that states words appearing within similar context possess similar meaning. Given this property, its applications are vast, as we will see later in the document. The most used and most popular method to get these vectors are neural networks, the cornerstone of Deep Learning.

The adequate building and implementation of said techniques and its joint assembly into a model that can solve the problem stated above has been the practical part of this project, alongside its evaluation. Hence, this work includes the development of a support ticket request automation with Natural Language Processing techniques.

Chapter 2

Problem Definition

Broadly speaking, the task to be performed by this project is to automatise support tickets. These tickets 5 are written by a customer, requesting of some service that Sourcesense may or may not offer (although most of the times they do because they already know what Sourcesense offers). The nature of the tickets can be very much diverse, and the ultimate goal is to extract keywords from all the incoming tickets. However, from the company they wanted to focus on the request tickets that refer to virtualization as they are the most common.

An example of a ticket request sentence would be:

"Goodmorning, I need two VMs with 4 cores and 16GB of RAM each. I also need them to have a 100GB SSD disk. Many thanks"

Hence, when a ticket arrives, it is usually a customer that asks for a virtual machine and its component specifics. A virtual machine (VM) is a virtual environment that functions as a virtual computer system, so they are a substitute for a real machine, as they provide functionality needed to execute entire operating systems. There are several uses of a VM such as the ability to run software the main operating system can't, or try out apps in a safe environment. Terminologically speaking, the operating system (OS) actually running on a customer's computer is called host and any operating systems running inside a virtual machine are called guests.

The components that Sourcesense needs to know in order to correctly process the request are mainly CPU cores, memory and disk storage:

- **CPU cores**: A processor (CPU as for Central Processing Unit) is the logic circuit integration that responds to and processes the basic instructions that drive a computer to execute. However, as we are talking about virtual machines, and to be accurate, we are precisely talking about vCPU's (virtual CPU) is a physical central processing unit (CPU) that is assigned to a virtual machine. By default, virtual machines are allocated one vCPU each, although Sourcesense allows customers to ask for how many the would like to have. To know how many vCPU's a customer needs is important, because processing time is billable, and adding more vCPUs doesn't automatically improve performance. For simplicity and for the remaining length of this document, vCPU's will be referred as CPU.

- **RAM memory**: Random Access Memory (RAM) is the temporary storage that goes away when the power turns off, and it is used for any task that requires fast access to computing resources; the more RAM a computing has the faster is able to run.

- **Disk storage**: Solid State Disk or Magnetic Storage Devices are the two types of disk storage that Sourcesense offers. SSD belongs to solid-state storage, a type of non-volatile computer storage that stores and retrieves digital information using only electronic circuits, without any involvement of moving mechanical parts. On the other hand, magnetic storage also falls to into non-volatile computer storage although it stores and retrieves the data using one or more magnetic rotating platters encased with magnetic material. They are the most economical forms of data storage.

A common structure, stablished by Sourcesense, of a ticket is the following:

Every ticket request, in order to be carried out successfully, must have the following criteria, or at least the mandatory fields

- **cpu**: <number of cores> (mandatory)

- **ram**: <number of gigabytes> (mandatory)

- **disk**: list of disks (mandatory); a disk has the following structure:

    1. size: <number of gigabytes> (mandatory)

2. type: <label whether is ssd, magnetic> (optional, defaults to magnetic)

- **name**: <string> (optional)

In addition, the quantitative fields (size, cpu, ram) may occasionally have a + sign in front of the number, to indicate that a relative increment is needed.

2.1 Approach

The main task can be defined in this terms: extract keywords from support request tickets; and once this is done, the automation of the tickets can be fully performed and streamlined. Also, this task can be expanded to extract practically any keyword from raw text and eventually process all incoming support tickets.

Given the nature of the task to be performed, it seems clear that the approach to be taken has to be related to text, text extraction, and hence the closest field related to Machine Learning is Natural Language Processing. However, it is not enough to know the field from where to start tackling the project, as it is very vast. Importantly, there is one main caveat that completely conditions all approaches: there is no dataset from which to train a model, because, although tickets arrive with certain frequency to the company, there isn't a database where to store them. The only ticket templates that the company could provide were a small sample of about 20 tickets in order to have a raw idea of the variability in terms of words that a ticket can have. Given this, approaches that rely on having a dataset from which to train a model have to be discarded and, possible reference cases such as of Tellina [3], a recently published paper that can translate raw sentences into bash code. That could have been a good baseline from which to build inspiration because the task in similar to ours, and it could be even more close if there would be a tickets dataset, since Tellina has built its own main corpus by pairing sentences and its bash comand translations. However, this corpus needed to be built by the hiring of some freelance coders that collected 12,609 text–command pairs in total, to which they built a neural model with them as input afterwards. Thinking it twice, although of course with much less resources than the authors of Tellina, it could have been feasible to build a dataset pairing request ticket sentences and its keywords. Nevertheless, as previously mentioned, Sourcesense doesn't keep track of any ticket, therefore they would have to be built and, just the intricacies of building tickets that can rely on a certain degree of resemblance to those of Sourcesense would give for a whole other project so it was found unnecessary.

Hence, the problem needs to be solved without a dataset from which a model could learn. In the quest of finding a suitable approach to build a method, it is usually useful to think about the most simple solution. Always in the NLP field, hard-coding a heuristic set of rules [4] that would retrieve the keywords seems too poor but it may be a good starting point. Combining well-known NLP tasks such as Named Entity Recognition and Dependency Parsing might be just-right to achieve a certain degree of efficiency. Nonetheless, a sentence requesting some service might have infinity of variations regarding semantics, grammar, morphology, sentence composition...Just the different options to name "RAM memory" are enough to make all the set of rules fail. And, considering how many things can be different, it becomes clear that this method is too simplistic and needs to be improved. But, for instance, it is a starting point, because as we know what kinds of words need to be found, we can establish just a few set of rules that will narrow down the search for posterior extraction. We also need to take into serious account that we are not just looking for a concrete word , but also to the number associated with this precise word. For example, when a customer asks for "RAM memory", the extractor needs to find and pop out how many gigabytes of RAM the customer desires. So, the two main techniques that helped achieve a general categorization of the words in the sentence were POS-Tagging and Dependency Parsing. By building a POS-Tagger, we are able to identify the grammatical category of each word and, as we know what words we are looking for in advance, we can at least infer that, if a word is a noun, it won't become a verb, so finding all words that are nouns scales down the search. Also Dependency Parsing, which finds the relationships between words in a sentence, is useful in order to find what quantity is associated to our keyword.

An example of the output of a POS-Tagger would be:

```
Hello -> compound -> INTJ
John -> compound -> PROPN
Smith -> npadvmod -> PROPN
, -> punct -> PUNCT
how -> advmod -> ADV
are -> ROOT -> AUX
you -> nsubj -> PRON
? -> punct -> PUNCT
```

Figure 1. Own POS-Tagging example on a random sentence

And, Dependency Parsing:

Nevertheless, just having a bunch of words that are possible candidates to become

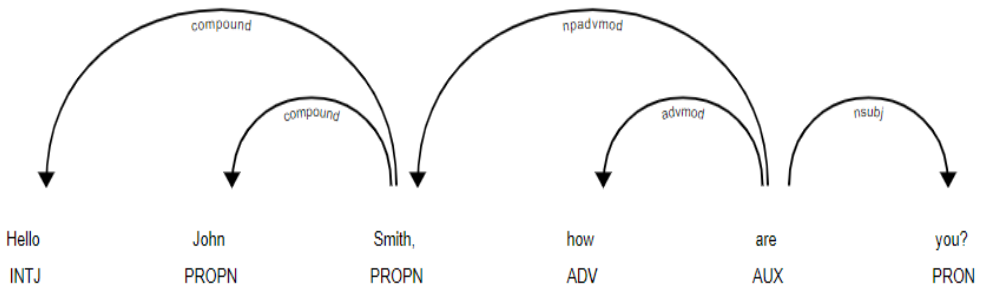

Figure 2. Dependency Parsing Example

the keyword and some syntactic relationships makes it possible to narrow down the search, but is barely nothing. The next step is to identify such word and extract it. To perform this task, we made use of Word Embeddings. They are exactly what we need because thanks to the embedding of a word we can find its closest words, always relying on the quality of the embedding. Hence, achieving embeddings that perform well in the task is key.

With all these ingredients, it is expected that the task can be solved with some guarantees. Given this, the Pipeline for which a ticket should go would be this way:

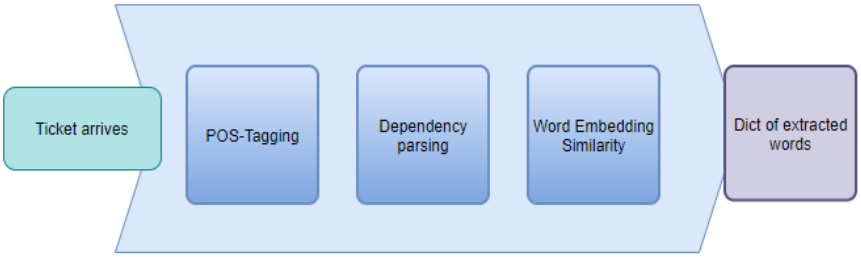

Figure 3. Overview

Working Environment

For the development of the above mentioned techniques Google Colab has been used. It is a free cloud service which supports Python and hence Python programming libraries. The choice has been mainly motivated by the weak computing power of the author's local computer, which would have been an issue when dealing with a lot of data when, for example, building the word embeddings.

Chapter 3

NLP Processes

As it was stated in the introduction, reducing the number of possible words that can become keywords can be beneficial, so for the completion of this goal, three different techniques methods have been performed. First of all, POS-Tagging for which a POS-Tagger was built, secondly, Dependency Parsing, performed using a famous built-in library such as Spacy and finally, Word Embeddings, built specifically for the task of these document although they might be useful for other applications. All methods are necessary for its own properties, as it will be shown.

3.1 Part-of-speech Tagging

3.1.1 Introduction

As previously mentioned, POS-Tagging essentially means reading a sentence and being able to identify what words act as nouns, pronouns, verbs, adverbs, and so on. All these word categories are referred to as the part of speech tags.

However, identifying part of speech tags is not trivial, because it is quite possible for a single word to have a different tags in different sentences based on different contexts. Hence, this is why it is absolutely advisable to rely on machine-based POS tagging.

Although typically POS tagging in itself may not be the solution to any particular NLP problem, it is something that is done as a pre-requisite to simplify a bigger problem, and it is commonly applied in tasks such as text to speech conversion or

word sense disambiguation. Also, it is useful to our project in order to develop a first cleanup of possible words on an incoming ticket that can potentially become keywords.

Types of POS-Taggers

Historically, there are two original types of POS-Taggers:

- **Rule-Based POS Taggers**: It is safe to say that Eric Brill's tagger [4] is the reference as far as Rule-based Taggers are. His paper explains one of the first and most widely used English POS-taggers, and it employs rule-based algorithms. So, typically they assign tags to words bt using its contextual information. Plainly, it is done by is done by analyzing the linguistic features of the word of interest, its preceding word, its following word, and other aspects. For example, if the preceding word is an article, then the word in question must be a noun,and all this information needs to be transformed into ground rules by coding them. Therefore, defining a set of rules manually is not efficient at all, but Brill's tagger though, defined that set of rules by going through the training data and optimizing each rule's tagging errors, so that the final set of rules are the ones with less tagging mistakes. Although rather simple, Brill's own conclusion indeed states that there is room for improvement and encourages researchers to continue his work, as he sees potential in it, in contrapostion on stochastic POS-Taggers, more accepted on the time of Brill's publication.

  The tags that are universally accepted by convention are:

TAG	WORD
ADJ	adjective
ADP	adposition
ADV	adverb
AUX	auxliary verb
CONJ	coordination conjunction
DET	determiner
INTJ	interjection
NOUN	noun
NUM	numeral
PART	particle
PRON	pronoun
PROPN	proper noun
PUNCT	punctuation
SCONJ	subordinating conjunction
SYM	symbol
VERB	verb
X	other

Table 1. Commonly used Tags

- **Stochastic POS Taggers**: they are basically any model which incorporates frequency or probability. The most simplistic stochastic taggers tag words based only on the probability that a word occurs with a particular tag, therefore the tag encountered most frequently in the training set with the word is the one assigned. However, as it doesn't take into account context too much, it can yield inadmissible sequences of tags such as article followed by a verb. A common approach is to define a generative model and maximize the probability of the hidden structure given the observed data. Typically, this is done using maximum-likelihood estimation (MLE) of the model parameter. Other methods might perform a Bayesian approach [5]

However, as the NLP field has been growing, more sophisticated methods have been found, specially with the rise of Deep Learning [6] [7] and, for this project it was used what is called a pre-trained BERT POS-Tagger. The reasons for it are mainly its particular technicalities and its accuracy on POS-tags, as it is shown next.

3.1.2 Model

BERT

The NLP field achieved an inflection point a few years ago with the development of Transfer Learning, and the development of pre-trained models. A pre-trained model is a model already created to solve a similar problem to the one of interest (in this, case POS-Tagging), because it has independently learned predictive relationships from training data, often using machine learning. So, instead of building a model from scratch, the pre-trained can be used as a starting point.

However, what's Bert? [8] To answer this question, we first need to distinguish between two different types of pre-trained language representations: context-free and context-based. The difference is if they take into consideration the words in context. As BERT is trained by taking into account both previous and next word, is context-based, as it makes use of the Transformer, an encoder-decoder network with attention mechanism that learns contextual relations between words (or sub-words) in a text. Encoder-decoder neural network is a type of neural network which is first use was machine translation. In short, English sentences are fed to the encoder, and the decoder outputs its translation, meanwhile an attention mechanism is a Deep Learning technique that helped improve the performance of neural machine translation applications by focusing on the more appropriate words regarding the input word at each time step. However, it proved to be useful for other deep learning neural tasks, so BERT can be used for a wide variety of language tasks.

So, to understand how our POS-Tagger works, we need to explain what a Transformer [9] is. As mentioned, on a high level, it is composed by an encoding and decoding components. An encoding component is formed by a Feed-Forward Network and a what it is called "Attention layer". The encoder's inputs first flow through the attention layer, which is a layer that helps the encoder look at other words in the sentence as it encodes a specific word. Thanks to this, the model it is able to find clues that can help lead to a better encoding for this word. To achieve this, each of the outputs of the encoder is sent it the decoder at each time step, therefore at each time step the decoder computes a weighted sum of all of the encoder outputs that is what determines the words most important to focus on. So, attention is the method the Transformer uses to increasing the understanding of a word by adding more useful relationships. Another key issue is that the inputs fed to the attention layer need to be embedding vectors, this is why it is necessary to convert the input tokens(words) into indexes first. Also, another thing that needs to be done is make sure the input sequence is formatted in the same way in which the BERT model was trained and, as BERT was trained on sequences that begin with a [CLS] token and uses also [PAD] and [UNK], so these tokens needed to be added.

Its outputs are fed to the feed-forward network, which in turn feeds the decoder. As it is known, the feed-forward neural network was the first and simplest type of artificial neural network created, and the information that it inputs just moves forward through the hidden nodes and finally to the output nodes. As the Transformer is usually built with 6 encoders, the inputs are just being sent to the next encoder every time.

As far as the decoder is concerned, its architecture differs just a bit from the encoder as it is added another attention layer whose technicalities differ just a bit from the encoder one. The mechanism however, is almost the same as the encoder. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results. The output of these is a stack of vectors, that need to be turned into probabilities that in the end will output a word. That is the job of a neural Linear layer and a Softmax layer. The Linear layer projects the outputs of the decoder into a large vector that contains the score assigned for the decoder to each word at a time step, and the softmax layer turns this scores into probabilities that add up to 1 by taking the exponents of each output and then normalize each number by the sum of those exponents. From these probabilities,

the highest one is chosen.The function is defined by (z are the scores)

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{3.1}$$

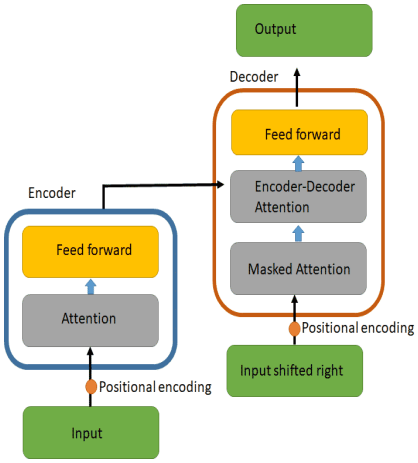

Figure 4. High Level Transformer Architecture

That's the structure of a standard Transformer for a machine neural translation task. However, Bert only needs to generate a language model; so as the embeddings are the only thing needed, all the decoder component it is not used. This means that the outputs of BERT are different, since they don't have to be passed through a decoder. However, there are more differences than just the outputs. For instance, as mentioned, the inputs of BERT consist of a sequence of tokens that needs to be numericalized and added some extra tokens([CLS]). The processing of the tokens of BERT includes two novelties in respect of the Tranformer. This is one of the reasons why BERT can be used for many different tasks. The first one is that BERT uses what it is called sentence embeddings. which are non other than a marker indicating to what of two sentences (A or B) a token belongs. This is done so that BERT can infer how likely is for a sentence to follow another sentence. Thanks to these markers, during training BERT receives as inputs pairs of sentences, in which half of the times the pairs are correct and the other half the pairs are just taken random from the corpus, and is able to learn how likely is to the second sentence (B) to follow the first sentence (A). Therefore, this means that BERT is pre-trained on a two-sentence classification task, which is useful for NLP tasks such as question answering. The second novelty that makes BERT so special is masking. BERT randomly masks 15% of the input words in the sentence and then it tries to predict them. And, while doing that it looks in both directions (bidirectional) and it becomes a context-based approach.

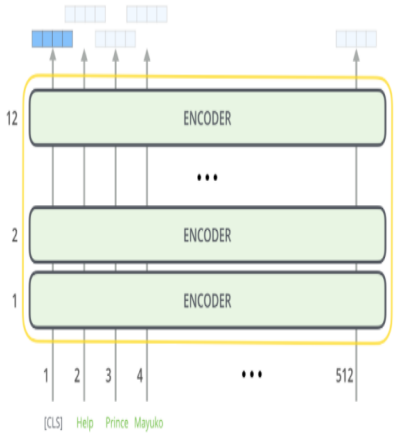

Figure 5. BERT Stack of Encoders

Now, thanks to BERT, we can fine-tune by adding a single layer on top of BERT's embeddings to predict the tag for each token in the input sequence and it is just right for the POS-Tagging task [10]. In Appendix can be found a the last single Bert Layer followed by the Linear Layer. 5

So once BERT is defined and the linear layer is added, just a few more things need to be defined. The Adam optimizer is the one used, with a learning rate of 5e-5 (0.00005), as it is one of the three values recommended in [8]. The loss function is created so that it calculates the accuracy of the tags that are correctly predicted excepting the support tokens ([CLS],[PAD],[UNK] which calculates our accuracy of predicting tags, ignoring predictions over padding tokens. The dataset used to train the model was UDPOS, from torch.datasets, which downloads and loads the Universal Dependencies Version 2 POS Tagged data.

WORD	TAG	TAG
Guerrillas	VERB	VBD
killed	VERB	DF
an	DET	DT
engineer	NOUN	NN
,	PUNCT	
Asi	PROPN	NNP
Alli	PROPN	NNP
,	PUNCT	
from	ADP	IN
Tikrit	PROPN	NNP
.	PUNCT	.

Figure 6. Extract of Dataset UDPOS

3.1.3 Results

So, after training the data for 5 epochs, the BERT POS-Tagger achieved very low train and validation loss (0.039, 0.280), while high train and validation accuracy

(0.9886, 0.9235). Here we can see the loss and accuracy curves. There are signs that the model might be overfitting a bit after the second epoch, since validation accuracy and loss stagnate although training accuracy and loss continue improving. However, these improvements are so tiny, and validation accuracy and test accuracy remain above 90 percent.

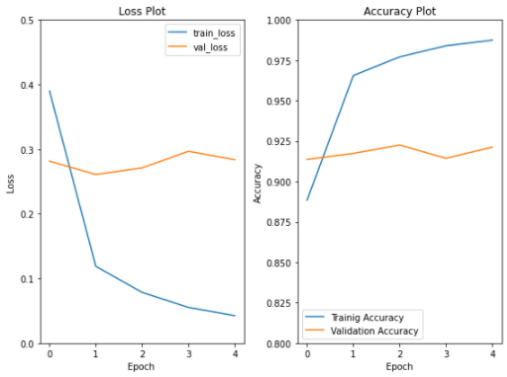

Figure 7. Loss and Accuracy Curves

3.1.4 Analysis

Here is a table regarding precision, recall and f1 score, as well as a confusion matrix, of every tag on evaluation.

1. Precision: tags classified correctly (True Positives) in a category (e.g. NOUN) divided by total predicted tags of such category (TP + False Positives).

$$\frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \tag{3.2}$$

2. Recall: tags correctly classified (True Positives) in a category (e.g. NOUN) divided by tags correctly classified (TP) plus tags that should have been classified as belonging to such category (False Negatives)

$$\frac{\text{True Positive}}{\text{True Positive} \ + \ \text{False Negative}} \tag{3.3}$$

3. F1 score: The F1 score is the harmonic mean of the precision and recall. The highest possible value of F1 is 1, indicating perfect precision and recall.

$$\text{F1} = 2 \times \frac{\text{Precision} \ * \ \text{Recall}}{\text{Precision} \ + \ \text{Recall}} \tag{3.4}$$

For a total of test 1583 tags, we can see that most of the tags are "NOUN", and that the overall accuracy for tags that are actually relevant, as it could be NOUNS or "NUM", is high above 90 percent. Regarding our task, that the POS-Tagger classifies with high precision "NOUN" is suitable. Some tags such as "INTJ" or X have not been almost tested because they are very rare, hence its metrics are 0 or close. It is interesting to see that most of the mistakes for "NOUN" are due to misclassification on "PROPN" (proper noun) , which is understandable and hence not very relevant. The model seems to perform very well on "DET" (determinants) and "PRON" (pronouns).

	Precision	Recall	F1-score	Support
NOUN	0.9567	0.9266	0.9414	286
PUNCT	0.9639	0.9816	0.9726	163
VERB	0.9739	0.9675	0.9707	154
PRON	0.9917	1	0.9959	120
ADP	0.9826	0.9941	0.9883	170
DET	1	0.9850	0.9924	133
PROPN	0.8343	0.9338	0.8812	151
ADJ	0.9381	0.9286	0.9333	98
AUX	0.9630	1	0.9811	78
ADV	0.9351	0.9730	0.9536	74
CCONJ	1	0.9623	0.9808	53
PART	0.9744	0.9500	0.9620	40
NUM	0.9048	0.6129	0.7308	31
SCONJ	0.9730	0.9474	0.9600	38
X	0	0	0	0
INTJ	0	0	0	1
SYM	1	0.33	0.5	3
				1593

Table 2. Classification Report of Tags

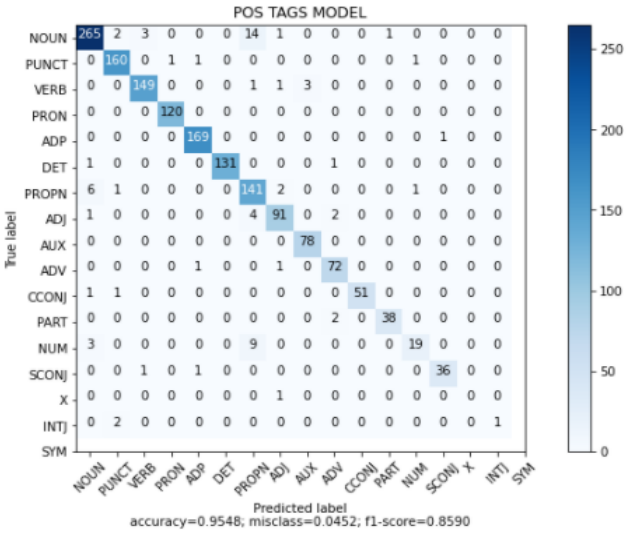

Figure 8. Confusion Matrix of Tags

3.2 Dependency Parsing

3.2.1 Parsing

Loosely speaking, parsing consists into processing written language into something more understandable in order to find meaning. The human brain does this on a daily basis when its reading. However, in the computer programming world, "parsing" is a task that consists into breaking a sentence into its components to describe its syntactic roles in order to gain some understanding. It is a wide used process in programming, since it means turning a piece of code written in a source language into some executable code that a computer can perform.

Typically, the sentence is broken down into some series of tokens, previously defined, that are language units by itself, for example, a "NOUN". Now, these language units can be of different nature and refer to different aspects of meaning, such as lexical or syntactical units. Hence, syntactic parsing algorithms specify how to recognize the strings of a language and assign each string one (or more) syntactic analyses, and a parser is a the program that carries this task out. The output of a parser is constructed in data structure which usually has a tree form, what is commonly known as a parsing tree. Generally, a parser works by first identifying the tokens as language units and then building the parsing tree.

3.2.2 Dependency Parsing

Dependency structures capture the relation between words in a sentence. Meanwhile POS-Tagging assigns to each word a label based on its grammatical category, it fails to acknowledge how are the words of a sentence related between them. For instance, POS-tagging can understand that "red" is an adjective and "carpet" is a noun, but it can't infer that this adjective "red" refers to "carpet". To construct the Text Extractor, it is needed more than just grammatics, since quantities of words are required (e.g. 2GB RAM). Hence Dependency Parsing is needed as to retrieve the relationship between each of the words in a sentence.

However, before talking about Spacy's Dependency Parser, we need to distinguish between 3 general methods of parsers:

- **Syntactic Parsing**: the goal is to provide a syntactic structure of a sentence. Basically, consists of defining a set of grammatical rules to reduce a sentence into sub-sentences from which joined together, they construct the original sentence. Usually these rules consists of phrase structure rules, for example

Determinant followed by Noun they constructs a Noun Phrase The common structure for this type of parsing is the parsing tree. In order to build the tree, two main approaches are defined: bottom-up or top-down. If we define as "S" the starting symbol for defining a the most general unit of a sentence (what would be the top of the tree), then top-down builds the tree iterating from this very first unit, meanwhile bottom-up constructs from the tree leaved till and finished when it reaches the top of the tree ("S").

- **Dependency Parsing**: parser in which every syntactic unit or constituent is connected to a head in terms of directed link. These connections are called dependencies, and are formed by a head word and a dependent word. The head is the central part of a syntactic constituent meanwhile the dependent is contingent to the head (e.g. A Noun(N) is the head of a Noun Phrase(NP)). The dependents can be either direct if they are in direct relationship with the head or indirect if they are related to the head through another dependent. The kind of relationship that bounds a head and a dependent is grammatical. So, it is defined by the grammatical role that the dependent plays to the head. A typical example can be a subject or a determiner. Aiming to have some sort of uniform criteria to define such grammatical relationships, linguists have developed complex taxonomies of relations that link two words in a very precise manner. The most popular and widely used for Dependency Parsing is The Universal Dependencies project, which provides a linguistically motivated and computationally useful set of relations. They can be applied in different languages as well. Hence, phrase structures rules do not play a direct role for dependency parsing, since the syntax of a sentence is described only in terms of the relationships that words (or lemmas) hold between them. For example, in "I" is subject of the root verb "am". An advantage of this approach is that it can be applied at the same to time to different languages because it doesn't take into account the order of words, opposing to syntactic parsing, which would have to define a rule that specifies all the different order of words combinations.

- **Semantic Parsing**: the aim of this type of parsing is transforming a sentence to a symbolic logical, formal representation. An example could be homonyms.

The parser used to perform parsing is the built-in Spacy Dependence Parser. The choice for using this already trained parser and not to build a custom one is because Dependency Parsers have achieved a reasonable amount of efficiency. Furthermore,

Clausal Argument Relations	Description
NSUBJ	Nominal Subject
DOBJ	Direct Object
IOBJ	Indirect Object
CCOMP	Clausal Complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numerical modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions and other markers
Other Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

Figure 9. Table of Simple Dependency Relations

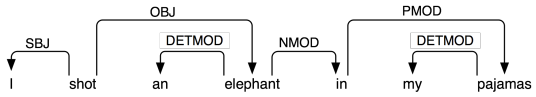

Figure 10. Example of Dependency Parsed Sentence

the task for which we require this type of parser is not of great specificity (just narrow down a set of keywords) and it was understood that there was no need to dramatically improve Dependency Parsers that already work well. However, for the sake of completeness and also to understand how the project makes use of NLP techniques, an explanation is in place.

3.2.3 Transition Based Parsers

Dependency Parsers can be understood as direct graphs, structures that link a set of vertices (V) with a set of arcs (A). Following this analogy, the vertices would be the words in a sentence while arcs would be the "arrows" that capture the grammatical relation of two words and join them together. Thus, when complete, it becomes a Dependency Tree. A Dependency Tree needs to satisfy three major issues. First, there a head (vertex) that acts as root so it has no incoming arcs; second, each vertex has at most one incoming arc; and third, there is only a unique path from the root head to each vertex. We also need to take into account that, if vertices are the set of words in a sentence, they also refer to punctuation signs.

Spacy's Dependency Parsers is what it is called a Transition Based Parser [11]. This type of parser starts in an initial configuration and, at each step, it asks a guide (oracle) to choose between one of several transitions (actions) that lead to new configurations. Parsing stops if the parser reaches a terminal configuration and when it does, it returns the dependency tree associated with the terminal configuration.

Transition Based Parsers are an advanced method from a Shift-Reduce Parse. A Shift-Reduce Parser maintains a parser configuration at any point when parsing the sentence with respect to a word, where a configuration(c) = (S, Q, A)

1. S: is a stack of nodes that are processed.

2. Q: is a queue of nodes that are yet to be processed.

3. A: is a set of arcs at some parsing point.

It works by taking as input a succession of tokens into the stack and the top two elements of the stack are matched against some pre-defined grammar rule (like Syntactic Parsing), and when this happens the associated symbol that represents the rule replaces the elements on the stack. Transition Based Parser develops further this approach by instead of replacing two words on the stack with a symbol, it defines a dependency relation. Indeed, these two words are the head and the dependent.

So, a TBP starts with an initial configuration of a stack of words, an input buffer of words to be sent to the stack, and a set of defined grammatical relations. Once the parsing process finishes, a final configuration is given in which all words have a dependency relation, the buffer is empty, the stack contains a single word and hence a dependency tree is built. To get to the final configuration, a series of so-called transitions need to be performed. These are:

1. Shift: push the next word in the buffer onto the stack.

2. Left-arc: add an arc from the topmost word on the stack, to the second-topmost word, so a relation is forged, and remove the second top-most word (the dependent) from the stack

3. Right-arc): add an arc from the second-topmost word on the stack, to the topmost word, so a relation is created, and remove the topmost word (also the dependent) from the stack.

Nevertheless, transitions need to be as accurate as possible so that the final dependency tree outputs the correct grammatical relationships. This is the job of the oracle (or guide) [12]. The oracle is the classifier that chooses which transition to apply at each parsing moment. As this task can be understood as classification, in

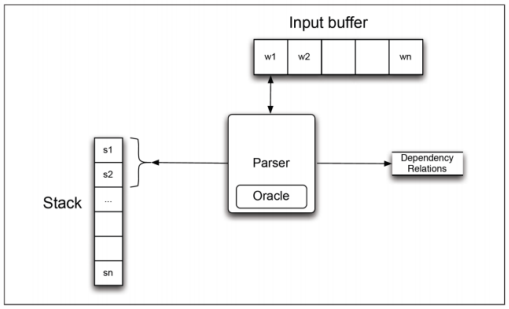

Figure 11. Overview of TBP

which the goal is to predict the next transition (class) given the current configuration, an oracle can be a trained as a supervised machine learning classifier. To do this, it would be necessary as training data some amount of configurations paired with the transitions applied to get to them. To have this for a lot of sentences is costly and there is no such data however, there exists the Dependency Treebanks. Treebanks are sets of annotated dependency parses for a language corpus. The Penn Treebank is the most used for English. So, since Treebanks provide a reference parse for a sentence, all the valid transitions can be inferred, and the oracle can be trained such that a transition is picked if it helps the current sentence parse to get to the reference parse, also called gold-standard. A training step would be for the oracle to choose over Left-transition or Right-Transition if this means that the built relationship concurs with the gold-standard. So, when all the Treebank data has been analyzed and all the configuration-transition pairs have been collected, it is time to train a classifier on them so that when the oracle receives an unseen sentence, it uses the classifier to predict what transition to apply.

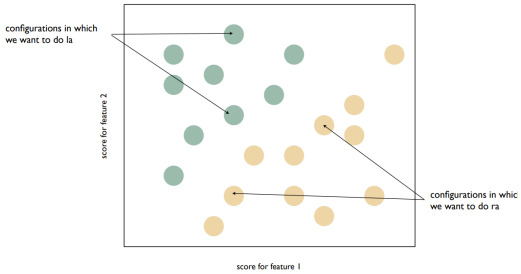

Figure 12. Graph of Configuration-Transition pairs

Despite that, for the oracle to be trained so it can predict what transitions to apply, a number of features derived from the configuration is needed. Features can be a very distinct nature, such as word lemmas or part-of-speech tags. In any case, they have to be relevant enough so that the classifier can rely on them when learning. At the end, every configuration can be described in a feature vector.

The most dominant approaches to training transition-based dependency parser classifiers have been multinomial logistic regression and support vector machines, although neural network, or deep learning are taking its place due to its superior benefits [13] [14].

However, the aforementioned model it's known as arc-standard although there are many other alternatives. Spacy uses a non-monotonic arc-eager model, which is more evolved than an arc-standard one. First of all, as its name implies arc-eager seeks for assigning heads to dependants as early as possible, as it is known that in general the longer a word has to wait to get assigned a head more likely a mistake can be made. This is achieved by slightly changing the transitions function and adding one more transition, known as Reduce. Now the Left and Right-arc transitions operate by establishing relations from the top of the stack and the top of the buffer and, as now the dependent is on the buffer, instead of removing the dependent from the stack it adds it to it so it may serve as head for following words. On the figure below it can be seen an example of how the parser works.

Transitions of an arc-eager transition model:

1. **Shift** : push the next word in the buffer onto the stack.

2. **Reduce**: pops the stack

3. **Left-arc**: adds an arc from the next input token if it is the head to the token on top of the stack and pops the stack.

4. **Right-arc**: adds an arc from the token on top of the stack if its the head to the next input token, and pushes the dependent word from the input buffer onto the stack.

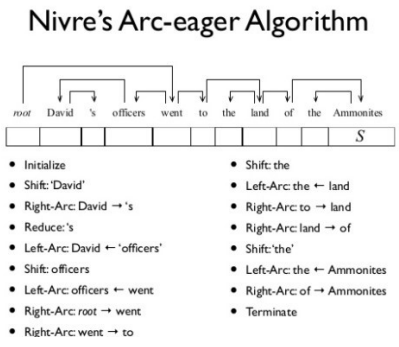

Figure 13. Arc-eager Transitions. Ref: Slideshare

Then, the non-monotonicity property term is referred to the parser's ability to repair earlier parsing mistakes by over-writing earlier parsing decisions. This done with the Unshift transition, which is applied only when the word on the top of the stack has no incoming arc and the buffer is empty. As explained above, this may have a harming effect, because more transitions increase the number of complete parse trees, and it delays the location in the input where the parser must ultimately commit a dependant to a particular head. However, [15] proves that it is still beneficial in accuracy terms.

Lastly, we need to talk about projectivity. A dependency tree is projective if for every arc in the tree, there is a directed path from the head of the arc to all words occurring between the head and the dependent. Other definition might be that it can be drawn with no crossing edges. While most of the sentences are projective, there might be some that are non-projective and might make the parser commit some mistakes, thus this needs to be accounted. Spacy's parser does so by projective applying pseudo-projective parsing, although for the purposes of these work is not relevant since most of the tickets are projective. Usually non-projective structures are useful for languages such as Czech [16], which is morphologically very rich which implies word order in a sentence is not subject to strict rules, or no rules at all.

Usually, the metrics to evaluate a Dependency Parser are:

1. Labelled Attachment Score (LAS): percentage of correct arcs relative to the gold standard reference parse.

2. Unlabeled Attachment Score (UAS) : percentage of correct head-dependant relationships, ignoring the dependency relation.

Spacy's is a non-monotonic arc-eager transition system, trained on the OntoNotes Release 5.0 Dataset that achieves a LAS of 0.90 and UAS of 0.92.

3.3 Word Embeddings

Just with Dependency Parser and POS-Tagger it is possible to narrow down a quest for a word: It can be done with what is called Information Extraction [17]. This method relies on sets of three words, called triples, which are formed with two words and a third that related them both. For example :(Tarantino, directed, Kill Bill). Given this sets of triples, it is possible to train a machine learning model so that it recognizes triples. In our case, it could be, for example (I, buy, RAM). However, as there is no data, the only way to approach this kind of task would be with a Rule-based approach, that is, define a set of syntactic rules and other grammatical properties of natural language that could be used to extract keywords. But there is no other choice than to discard this approach because an incoming ticket may have infinite variations if we think of synonyms, phrase structure or punctuation, hence defining rules that comprise all these options is an unfeasible task. Hence, another NLP technique which its properties fit perfectly with our purpose are Word Embeddings.

3.3.1 What are WE

Plainly, Word embeddings are numerical representations of words. The simplest way in which they can be represented is by using on-hot vectors. One-hot vectors allows to represent categorical variables as binary vectors, in which each integer value assigned to the variable is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1. As a visual example, if there are two words, "book" and "shelf", the first will be represented as [1,0] and the second [0,1]. On a large scale, however, obvious problems may arise. For instance, if we project the one-hot encoding procedure to thousands of words, not only the dimensions of the vector would explode but also it can turn into an sparsity issue due to the huge amount of 0s that there would be, causing the model to generalize poorly. However, what's really an issue for the aim of the project is that only using one-hot vectors doesn't take context into account. This is concept of word similarity is key and thus, a model that finds similarity between words is needed.

The solutions comes from what's called Iteration Based methods, which is models that try to learn the probability of a word occurring one iteration at a time and encode the likelihood of a word given its context so there is similarity between words. There are two main methods for creating Word Embeddings, which are Common Bag of Words (CBOW) and Skip Gram Model (SGM). Both methods need to give thanks the famous [18], which first came up with the notion of distributional hypothesis,

that builds on the intuition that words occurring in similar contexts have as well similar semantic meaning. This idea will be exploded when building the custom word embeddings.

But first, let's define CBOW and SGM. The idea behind CBOW is that it predicts if a word is likely to appear given its context. In the sentence "The cook baked a cake", it could be tried to predict "baked" from its context words "The,"cook","a","cake". In this sense, when using CBOW and there are large documents in the dataset, a context word parameter "C" can be stablished. This neural network usually has three layers; an input layer, a hidden layer and an output layer, and two sets of weights. These weights are the ones learned by the CBOW model in form of matrix, and the weights between the hidden layer and the output layer are the ones used for a word numerical representation, and thus the word embeddings. Now, CBOW makes use of one-hot vectors in order to encode words. If the hyperparameter "C" is 2, there will be two one-hot vectors as inputs in matrix form for just one target word. For the model to learn the weights, the one-hot vectors of the context words are all stacked together in a matrix form and are passed to the aforementioned neural network. The network optimized the probability of the word occurring with a negative log-likelihood loss function

$$H(\hat{y}, y) = -\sum_{j=1}^{|V|} y_j \log(\hat{y}_j) \tag{3.5}$$

and gradient descent. A negative log-likelihood function is suitable for this task because it provides a measure of distance of how far is the probability generated to the true probability. To calculate the probability is done with a softmax function on top of the output layer.

On the other hand, the SGM is similar to the CBOW but its conceptual framework is the opposite, since it tries to predict context given a center word. Hence, the output vector of Skip Gram will be two target variables for just one input word; the loss function is the same and it is trained with gradient descent (Stochastic Gradient Descent) as well, although in this case it will be optimizing two errors, one per each word.

3.3.2 Data gathering Description

Selecting the data to be fed to the model to build the embeddings is a delicate choice. There are a lot of built-in corpuses ready to use, of famous Libraries such as

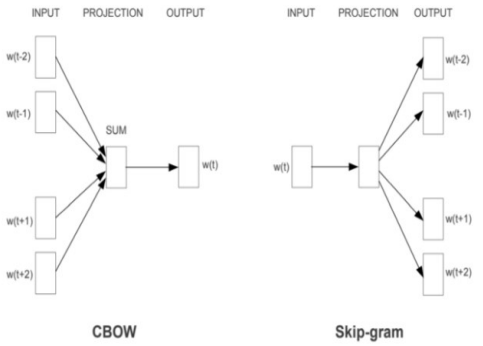

Figure 14. CBOW vs SGM

Gensim or NLTK. However, the first drawback of these corpuses is that, besides that some of them are so large that the require too much memory, they may not contain data specifically related to computing. In other words, their domain is too general. Several researchers have studied what is better, if a large corpus or a domain specific one. For example, [19] presents the applicability of word2vec embedding model to the task of extracting similar words from small, domain-specific data. Results show that the specificity of the corpus has much more influence on word2vec results than its size, which leads them to conclude that when the goal is to automatically detect similar words that are domain specific, it is better to have a corpus that correctly represents the use of those specific words more than to have huge amounts of data unrelated to the targeted language. In addition, [20] demonstrates the advantage of specializing embeddings for the tasks of genuine similarity and relatedness. To do so, specializing for similarity is achieved by learning from both a corpus and a thesaurus and two specific two retrofitting methods for learning embeddings. Their final conclusion is that specialized embeddings outperform standard embeddings by a large margin on instrinsic similarity and relatedness evaluations. Following this line of work, [21] states that, after evaluating six text corpora of varying size and similarity to the target corpus, it was found that combining multiple corpora, is the safest option for choosing embeddings.

Therefore, the approach for building the dataset relies on gathering data of the same domain of our keywords and data from its thesaurus environment. Originally, thesaurus is a type of dictionary that aggregates “clusters” of words with similar meanings grouped together. Hence the data to feed to the embedding model has been collected from Wikipedia articles related to our words of interest. As previously stated, the idea behind this choice is that we want to create a corpus that specifically contains words that are in a context related to computer data, and also, we don’t

want to run into memory issues when working with those such big corpus. Since the interest is on semantic similarity, it would be beneficial that, for example, the word "disk", is close to "storage" rather than "music", which means that it is far more efficient to build our own corpus and train it. Wikipedia choice is based on the fact that Wikipedia articles are grouped into categories that join together pages on similar subjects, and this distribution makes the search for words that have similar context suitable. Also, inside categories Wikipedia has also defined subcategories. So taking advantage of Wikipedia's distribution of its articles, categories and subcategories of topics related to the keywords can be extracted quite easily. To do this, it was used the WikiExtractor to download all the articles and then uploaded to Google Colab for its processing. An extract of the categories can be found in the Appendix 5.

In the end, 6.311 articles were used as corpus to train the embeddings, from which there 30.194 sentences and a total of 360.075 words.

For visualization purposes, here are some graphics that illustrate an overview of the dataset, and below the same but for some already made corpus, such as NLTK "Webtext Corpus", a corpus made up of content from a Firefox discussion forum, conversations overheard in New York, the movie script of Pirates of the Caribbean, personal advertisements, and wine reviews. This corpus contains 396.733 words, although as it can be seen in the graph, they are not related to the topic of interest and also there are just a few occurrences of the keywords.

The reasons stated above also explain why pre-trained embedding vectors from libraries such as GloVe should be rejected.

3.3.3 Data preprocessing

In order that the data can be fed to the model, it needs to be preprocessed. The preprocessing pipeline contains several well-known NLP techniques. As explained. It works as follows:

1. Tokenize the sentences. By applying sentence tokenization the articles (that contain many sentences) are split into individual sentences.

2. Removal of special characters such as punctuation marks.

3. Removal of single characters, like "a".

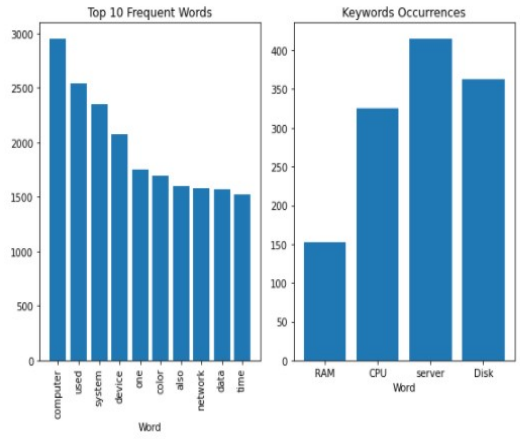

Figure 15. Hand-made Corpus Data Description

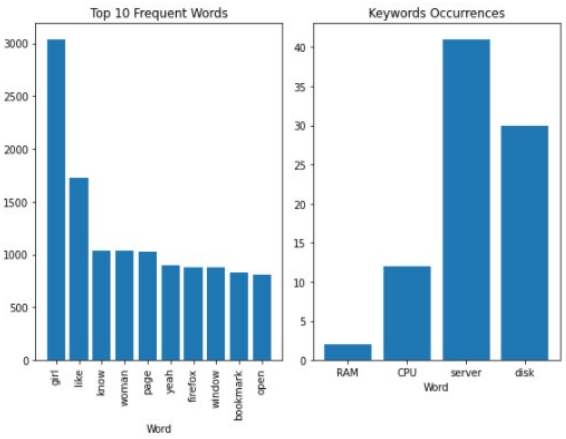

Figure 16. Webtext Corpus Data Description

4. Removal of extra white space by replacing with just single space.

5. Removal of digits.

6. Removal of Wikipedia articles headers.

7. Convert all text to lowercase.

8. Lemmatize each word, which means to try to reduce a given word to its root word, called its lemma. This step is done with NLTK NLP library.

9. Removal of stopwords. Stopwords are defined as words that don't add much meaning to a sentence and can be ignored without any regards. For example, the words like "the", "I"...

3.3.4 Model Selection Hyperparameters

To build the word embeddings we make use of FastText built-in library. Using a built-in library allows us to implement better embeddings since we are not constrained by computing resources and also because FastText [22] implements some developments to the classic [18] that is worth taking into account.

FastText Library

As previously explained, Skip-Gram and CBOW are the main methods to build embeddings. Whatever great they are, they still face some drawbacks. The first one is concerned with Out-of-vocabulary words. Plainly, words that are not seen in the training corpus doesn't have an embedding since both models create embeddings for concrete words. The second flaw that can be improved is that words that have the same root. for example, "process" and "processor", are just learned by its context, hence in the unlikely but still possible case these that two words have dissimilar contexts, its embeddings will not be as similar as could be. To solve these issues, FastText algorithm generate character n-grams for each word. Character n-grams are groupings of some characters of a word. For example in "basketball", 3 character n-grams would be "bas", "ask" and so on. the number for generating the n-grams is a sliding window parameter. As it is obvious, this approach might consume a lot memory, FastText uses hash values that are assigned to each n-gram, where each hash value gets mapped to a bucket, in order to develop some control on the memory expenditure. The downside of this method is that, as the number of n-grams might be higher than the bucket size, it may happen that too many hashes get mapped to the same bucket index, causing collisions between n-grams. However, as the memory requirements are usually very relevant, this necessity becomes prioritary. Hence, thanks to this, out-of-vocabulary words can be fetched because as there are subwords, when a new word appears the average vectorized representation of its constituents becomes the embedding. This is just great for our purposes since, for example, it might happen that a client refers to a "server" (1 word) as "virtual machine" (2 words).

Another key development in FastText is Negative Sampling. The idea behind it is that by optimizing a loss function that contains unnatural words that normally under no circumstances would be around the context of center word and words that indeed are around the center word. This way these unnatural words, called negative samples, get a low probability of ever occurring around a center word and these causes that the distance from the center words increases. This aforementioned

sentences can be generated on the fly by randomly sampling from the own words of the dataset, and the optimization is carried by taking the dot product of the center words and its context words with a sigmoid function and applying Stochastic Gradient Descent to compute the optimization so that negative samples get further away the center word. The objective function is:

$$\log \sigma \left(u_{c-m+j}^T \cdot v_c \right) + \sum_{k=1}^{K} \log \sigma \left(-\tilde{u}_k^T \cdot v_c \right) \tag{3.6}$$

where the first term contains the dot product and the sigmoid applied to the context words and the second term the same computations applied to the negative samples, which are sampled with some established probability.

According to the original paper [22], it is found that Skip-Gram works well with small datasets, and can better represent less frequent words, meanwhile CBOW is trained faster and can better represent more frequent words. For our project, based on this we could say that skip-gram would work better since the dataset to build the embeddings will be of a rather normal size and the words to find are not exactly of daily use, and some rather infrequent words might come up due to the dataset being pretty technical and specific ; therefore it was the chosen one.

The hyperparameters that fastText allows to set are:

1. **Embedding size**: size of the embedding vector. It has been set to 60.

2. **Window size**: Words of context that surround a center word to take, before and after. Set to 40 words.

3. **Minimum of words**: Minnimum number of occurrences that a word has to have in the training corpus so that an embedding is generated. Since our corpus is highly specific, the approach taken implies that if a word does not appear a decent amount of times, in this case 30 times, it is not relevant. This is done to discard low frequency words.

4. **Down-sampling ratio**: Downsampling ratio applied to the highest frequency word. This technique has been proven beneficial for word embeddings on skip-gram. This is done to Word to lessen the impact of high frequency words

or increase the relative importance of words closer to the center of a context window. It is set to $1e-2$

5. **Iterations**: number of iterations. Set to 100 iterations.

The model run on CPU times 25min 37s to train the embeddings.

3.3.5 Visualization of Word Embeddings

Here is a complete visualization of the Word Embeddings. First of all, will take the standard word "computer" and see its vectorized representation.

```
array([-0.13858508,  0.4845619 , -0.03507609,  0.37562862, -0.24291725,
       -0.37549233, -0.19930334,  0.07452878,  0.2244361 , -0.17655514,
        0.31201938,  0.4623441 ,  0.11916021,  0.05054465, -0.16273364,
        0.2984102 , -0.49163136, -0.11033346,  0.02095287,  0.04592944,
       -0.1511742 , -0.22605333,  0.12542349,  0.17549516,  0.264852  ,
       -0.20391332,  0.26949126,  0.15250032,  0.5105884 , -0.22215691,
        0.12716864,  0.37917438,  0.10848423,  0.416552  , -0.05569207,
       -0.04298532,  0.19371293,  0.24668182,  0.21767098, -0.6442474 ,
        0.18962319,  0.05593148,  0.10028459, -0.17263354,  0.3694023 ,
        0.07023641,  0.35033968, -0.12617318,  0.483398  , -0.10430112,
       -0.02451986,  0.3973285 ,  0.05596754, -0.13711637, -0.08453146,
        0.45497072, -0.21831672,  0.1230672 ,  0.4599974 ,  0.04430065]
```

Figure 17. Vectorized representation

Each word in the vocabulary is representation by a vector of size 60. Given this, the similarity between two vectors can be calculated with the cosine similarity. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space. The formula is defined by

$$\text{Cos}\,\theta = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\|\vec{b}\|} = \frac{\sum_1^n a_i b_i}{\sqrt{\sum_1^n a_i^2}\sqrt{\sum_1^n b_i^2}} \tag{3.7}$$

where the numerator is the dot product of two vectors and the denominator the cross-product of the length of two vectors. Given this, let's see what are the 10 most similar words for the keywords along with its similarity scores. It can be appreciated that in fact the embeddings are quite precise, since for example "processor" is the most similar word to "cpu" or "memory" the most similar to "ram". And having seen this and related to what was exposed before we can see just at sight that these embeddings make more sense to our task than, for example, GloVe pretrained embeddings. Let's compare the word "disk". For our purposes, it is much more relevant than it is close to "storage" rather than for example, "ghz".

	SERVER		CPU		RAM		DISK	
	Word	Similarity	Word	Similarity	Word	Similarity	Word	Similarity
1	client	0.79	processor	0.75	memory	0.67	drive	0.83
2	host	0.7	processing	0.7	chip	0.63	hard	0.74
3	web	0.68	chip	0.62	rom	0.61	floppy	0.68
4	network	0.628	interrupt	0.597	vic	0.59	file	0.63
5	dns	0.627	gpus	0.593	stored	0.56	flash	0.627
6	internet	0.622	gpu	0.5932	onboard	0.55	storage	0.6255
7	service	0.621	memory	0.580	video	0.546	tape	0.6251
8	fabric	0.621	borad	0.572	graphic	0.544	sector	0.618
9	node	0.61	board	0.561	williams	0.54	block	0.599
10	virtualization	0.6	intel	0.56	ultrasound	0.53	write	0.58

Table 3. Top10 similarities

DISK			
GloVe		Model	
Word	Similarity	Word	Similarity
hdtv	0.86	drive	0.83
disc	0.85	hard	0.74
photobook	0.82	floppy	0.68
widescreer	0.82	file	0.63
dual	0.81	flash	0.627
lcd	0.81	storage	0.6255
projector	0.807	tape	0.6251
panel	0.806	sector	0.618
mhz	0.801	block	0.599
ghz	0.79	write	0.58

Table 4. Pretrained vs Model

More evidence can be still found if we check the similarity for "ram" with "memory". This example is quite ilustrative because this pair of words need to be as attached as possible since the customer support tickets specifical ask for this. So, GloVe pretrained embeddings provide a similarity of just 0.37, meanwhile the custom embeddings one of 0.67. Also, if we take the word "virtual machine", GloVe doesn't have any embedding for, meanwhile this model does.

Now let's see how the embeddings are distributed. Firstly, let's see how the keywords occupy the space, concretely a 3d space. Secondly, let's see a general view of some embeddings. Here there is a general vision of the embeddings alongside one of the keywords embeddings and its 15 closest pairs.

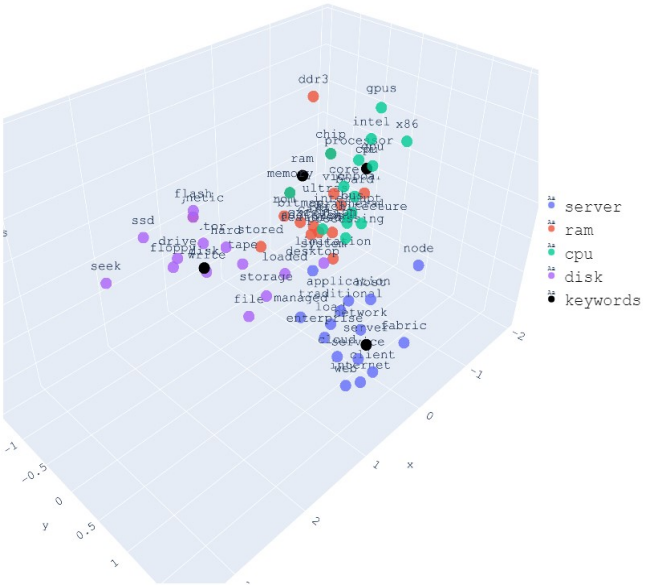

Figure 18. Keywords representations in 3d space

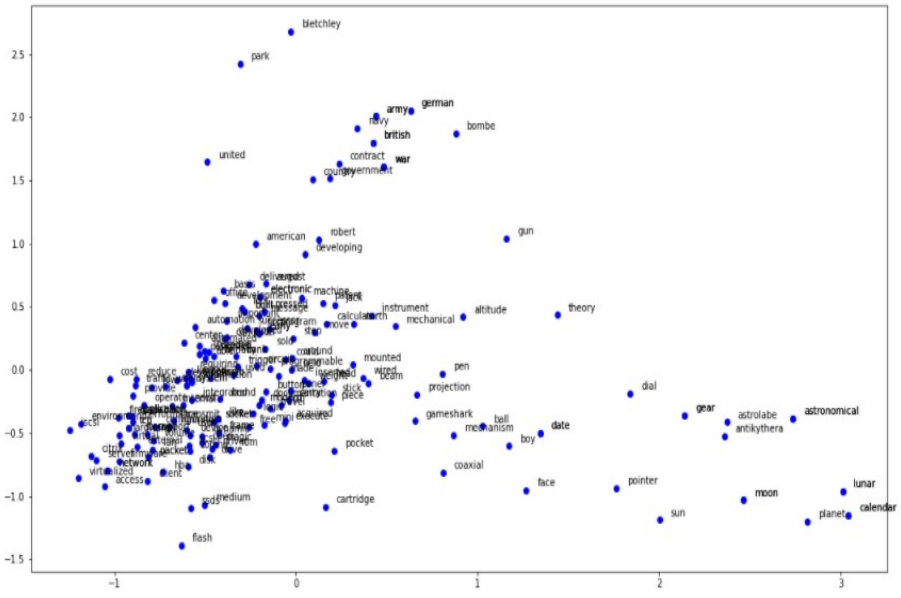

Figure 19. General overview in 2d space

The first plot shows that despite all the keywords being relatively close to each other, there can be spotted different categories of words according to which keyword they are related. It can be appreciated that words related to "cpu" and "ram" mesh quite close in the space, implying a tight bounding with these two concepts. Interestingly enough, words that are the center of the plot are the ones that represent general

concepts in computing, such as "desktop" (purple) or "system" (red). The words that lie further are in fact intuitively not very related to the ones of interest, such as "seek" (purple) or "node" (blue).

The second plot shows a more general view of the embeddings, with more words. The cloud of words at the bottom-left of the plot are embeddings related to computing, such as "bus", "processor" or "board", and as we scope out of the cloud more topics appear. For example, on the further bottom-right there exists a topic that might be called "cosmos" with embeddings for "planet", or "lunar". Also on the center top, words totally unrelated to computing such as "park" or "army", find its position.

These plots have been generated applying Principal Component Analysis (PCA). PCA is a technique used for dimensionality reduction. It is useful when the vector space is very large consequence of being high.dimensional, as in the case of the embeddings, which its vectors have length 60. PCA is the most used algorithm for dimensionality reduction. It works by first identifying the hyperplane that lies closer to the data and then projects the data onto it. The hyperplane chosen is the one that preserves the largest amount of variance, which is important so that the projections maintain its original distance. To find that hyperplane, the principal components have to be found. The principal components are the axis that account for more variance, and can be found with the matrix factorization technique called Singular Value Decomposition. Once they have been found, the dimensions can be scaled down by projecting it to the hyperplane defined by the components, which allows large vectors into a lower space and hence the embeddings can be visualized accurately.

Chapter 4

Pipeline

The end goal of this work is to generate a dictionary that contains the quantity for each ticket keyword request. To visualize how this process is done, we will describe all the steps that happen when there is an incoming customer support ticket. The keywords that were selected to represent the fields in the ticket to be retrieved were "virtual machine", "processor", "ram" and "disk". The selection of this words was done by just because there were the most common in the request tickets sample.

So to retrieve each keyword in case there is one, at first the ticket goes through the POS-Tagger, that labels all the words accordingly. After, a function finds and returns all the words that fall into the NOUN category. Then, all the nouns are preprocessed in the same way the embeddings were, in order that they can be recognized. At this point, the candidate nouns are passed to another function that first of all checks if the candidate word has an embedding in the vocabulary and after computes the similarity of every candidate word with a specific keyword established as a parameter. Once all the similarities are computed, it outputs the word that has the highest score, and for now, this keyword is retrieved.

A more complex procedure has to be made for retrieving the number associated to the keyword (e.g GB RAM). The initial step is to apply the Spacy NLP Dependency Parsing package to the ticket for later use. After that, the ticket request gets to a sentencizer. A sentencizer is just a function that splits a set of sentences into single sentences. Then, as we assume that one of the sentences contains the information associated to a keyword that we want to retrieve, each word of the sentence is analyzed with the similarity measure, always provided that an embedding can be

formed and it is in the embedding vocabulary. Then the sentence that has a word whose distance is closer to the embedding of the keyword is more likely to contain the keyword, hence is outputted. After, an intermediate step is performed as it was disrupting the correct functioning of the process, by adding noise. This step is to remove whatever adverb accompanies a noun, which usually is the word "of". For example, "X number of servers". Immediately after, Dependency Parsing is performed on the sentence and, all the terms that fall into the category of "nummod", which means numeral that accompanies a noun, are selected. A feature that can only be applied with Dependency Parsing is to find the relation head-dependant. Hence, as "nummod" points to a noun, this noun is also selected. Hence now there a few tuples that contain a number and a noun, which are to be considered the candidates. Therefore, it is time to find the distance between this candidates and the keyword and to retrieve the word which has the maximum.

It is important to mention that it may happen that the word with the highest score is a word that is totally unrelated with the keyword, hence this word has to successfully reach a concrete similarity threshold to be a valid output. Establishing this threshold is not trivial, since setting a threshold too high might incur in discarding words that in fact are partial synonyms to the keyword, and setting it too low might incur in outputting wrong words. Given that we also have just a few tickets that Sourcesense sent, it was decided to artificially generate more tickets 5 and perform grid-search on them. To generate those tickets, some of them were generated using templates that generate different ticket structures, and the keywords were replaced with words from a thesaurus 5, meanwhile other were hand-written generated based on the original ones. The grid-search was performed with several thresholds ranging from 0.1 to 0.9. As each keyword has its own retrieving function, 4 grid-searches were done. The accuracy metric to determine what threshold works the best is the percentage of correct retrieved words. Once the best threshold is retrieved it is applied in different steps of the process so that misleading words get discarded.

The aforementioned process is the standard procedure when retrieving a keyword, although for some keywords there are specifications that had to be made. For example, in the case of "servers", it was established that when the word appears without a number, this number is set to one. Or, when retrieving "disk" and "ram", the threshold had to forcefully take into account "gb" (as for gigabytes) since sometimes "disk" and "ram" are omitted.

To sum up, there are four different keyword retrieval functions, that when joined

together, output a dictionary with every keyword and the quantity associated to each. If a keyword is not present, it remains blank. The total time for processing a ticket is in between one or two seconds on average.

Here is one view of an example of an standard ticket keyword extraction and an overview of the pipeline. An extract of some keyword extracted tickets can be found in the appendix 5.

'Hi, I need: 3 servers with: 2 CPUs; 24GB RAM; 50GB of disk.'

{'cores': ['2', 'cpu'], 'disk': ['50', 'disk'], 'ram': ['24','ram'], 'server': ['3','server']}

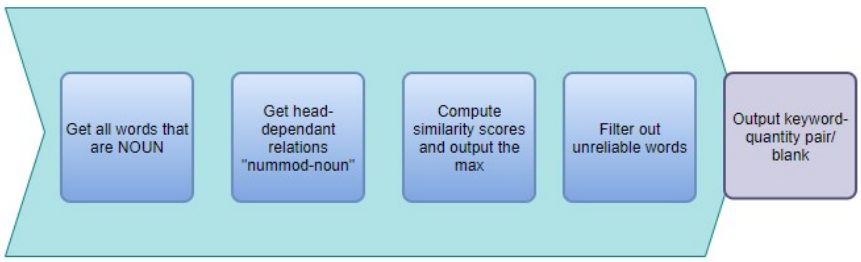

Figure 20. Pipeline overview

4.0.1 Results

Although there isn't a dataset for which the model can tested, here is an attempt to shine some light in how the model would work on the daily basis. It is important to note that, if the model achieves an accuracy of let's say 50 percent, this means that 50 out of 100 could be automatically managed, which may still save a lot of man hours.

The testing process has been carried out with the artificial tickets. To compute a measure of accuracy, the expected correct extraction needed to be generated by hand. Hence, for every keyword, a pair of (keyword, quantity) is the correct extraction. Thus, accuracy consists of the percentage of correct keywords retrievals, were retrieving blank is an option to be considered too.

Nevertheless, since it is impossible to build a dataset that can take into account all the variability that several persons can employ when writing a request ticket, just an approximation of some conclusions can be attempted. Despite that fact,

several things can be known for sure. The model works well with words that have a representation, as well as sentence with usual sentence structures and general knowledge words. Also the variability for which the model can incur in errors is hard to precise, although it can be said the complexity of the model itself provides automatic filters, the most important of those being the similarity threshold. For instance, even if a POS-tag or a head-dependant pair is wrongly classified, then the word embeddings will discard it anyway. It could happen, and this is a source for errors, that a tag wrongly classified then has a high similarity with an embedding, although this is not to be expected often. Lastly, it may cause some errors words that due to a wrongly set similarity threshold, which discards words that are in fact close to the keyword. Also, might happen that there is a quantity but there isn't an output keyword. In this case, the quantity is discarded.

The model was tested on a small sample of 50 tickets. The accuracy metric consist of the average of the accuracy for every keyword and its quantity retrieved correctly, which was 77.13 percent. Obviously, this accuracy is directly related with how the test is performed, since all the sentences follow a rather normal structure and there aren't mistakes or typos. Nevertheless, on a daily basis, the data that it generates could be recorded in order to make improvements, as adding more word categories to the embeddings or add more than one head-dependant valid relationships.

Chapter 5

Conclusions

One of the first conclusions that have been reached is that while trying to perform a keyword extraction, it is important to approach the problem as a process in which the final keyword is retrieved because all other words have been discarded. Hence the task becomes one of successively narrowing down the set of words that can become keywords. To perform this process, it is best to employ Natural Language Processing techniques that classify words according to a criteria for then selecting words belonging to one or more criteria.

As well as other machine learning problems, NLP techniques are becoming more straightforward to solve thanks to deep learning, which increases its overall performance. However, an important thing to keep in mind when approaching a keyword extraction task is that each case is special, there isn't an universal recipe. In this sense, extracting keywords really depends on how such techniques are implemented. The combination that this project employed was to use a Pos-tagger, a Dependency Parser and to build word embeddings, although keyword extraction might be performed in different ways depending what words are needed to be extracted. For example, if the words are entities, Named Entity Recognition (NER) would be in place. In this sense, for future work regarding this project, NER could be applied in order to retrieve the name of a server that has been already assigned to a client and there is a reference of it in a support ticket.

Regarding said techniques, if we speak of the case of a POS-Tagger, the development of open source deep learning models and transfer learning has made possible for pos-taggers to achieve high accuracy in most of the tags, becoming very reliable.

Specifically, the particularities of BERT and Transformers allow to train a Pos-tagger that performs well. A similar case is regarded to Dependency Parsing, although the influence of deep learning is not as determinant as in pos-taggers.

Nevertheless, while Pos-tagging and Dependency Parsing contribute to narrow down a set of word on a large scale, is the application of Word Embeddings what makes a model to be able to choose between semantically similar words, hence are indubitably the most important technique to implement. Doing this project it has been proved that it is better to build custom word embeddings rather than use some pre-trained models, specifically if the word embeddings are words that share a common technical field. For this reason, it can be stated that it is feasible to retrieve any desired words provided that there are word embeddings that capture well the semantic relationships of the words of interest. To achieve that, gathering as much data as possible related to a specific topic seems the best choice, as well as, using Fasttext algorithm because provided out-of-vocabulary embeddings. Despite that, evaluation of word embeddings is difficult to assess on a keyword extraction task if there isn't data. This would be the main problem that a keyword extractor has to face, although the best way to evaluate might be simply putting it to practice in a real-world scenario on a test mode. Generating some artificial data would be another option, although it seems rather complex to build a dataset that can resemble real-world data accurately.

Finally, to ensemble all of the techniques into one pipeline where the functionalities of each are exploited can be done smoothly since all the techniques approach different challenges that don't overlap, so the transition of a ticket through them comes out naturally.

It is worth stating that the next steps once the keyword extraction model is done, as mentioned at the beginning of the document, are meant to be a chatbot that demands the clients to confirm if what the model outputs is correct to later pass the information to a program that simply executes the task of creating the virtual machines, instead of clicking in Amazon Web Service interface. For this, an structured text containing all the information regarding the keywords would need to be sent, exactly what the keyword extractor provides.

As a final conclusion, since extracting keywords has been carried out successfully for a not so uncommon type of ticket, it can be inferred that automating processes with a keyword extractor can be applied with success for more cases.

Bibliography

[1] Wikipedia. Moore's law, 2015 [Online]. Available http://https://en.wikipedia.org/wiki/Moore%27s_law.

[2] Sourcesense. About us, 2021 [Online]. Available: http://https://www.sourcesense.com/about-us.

[3] Deric Pang Kevin Vu Luke Zelemoyer Michael D. Ernst Xi Victoria Lin, Chenglong Wang. Program synthesis from natural language using recurrent neural networks. 2020.

[4] Eric Brill. A simple rule-based part of speech tagger. 1993.

[5] Sharon Goldwater and Tom Griffiths. A fully Bayesian approach to unsupervised part-of-speech tagging. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 744–751, 2007.

[6] J.A. Perez-Ortiz and M.L. Forcada. Part-of-speech tagging with recurrent neural networks. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, volume 3, pages 1588–1592 vol.3, 2001.

[7] Sasikumar M. Raj Nath Patel, Prakasj B. Pimpale. Recurrent neural network based part-of-speech tagger for code-mixed social media text. 2007.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational

Linguistics, 2019.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[10] Melvin Johnson Naveen Arivazhagan Xin Li Henry Tsai, Jason Riesa and Amelia Archer. Small and practical bert models for sequence labeling. pages 3632–3636, 2019.

[11] Xun Zhang Yantao Du Weiwei Sun Xiaojun Wan. Transition-based parsing for deep dependency structures. 2016.

[12] Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, pages 959–976. The COLING 2012 Organizing Committee, December 2012.

[13] Christopher Manning. Chen, Danqi. A fast and accurate dependency parser using neural networks. pages 750–750, 2014.

[14] Xinchi Chen, Yaqian Zhou, Chenxi Zhu, Xipeng Qiu, and Xuanjing Huang. Transition-based dependency parsing using two heterogeneous gated recursive neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1879–1889, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[15] Matthew Honnibal and Mark Johnson. An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1378. Association for Computational Linguistics, 2015.

[16] Joakim Nivre and Jens Nilsson. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 99–106. Association for Computational Linguistics, 2005.

[17] Sonit Singh. Natural language processing for information extraction, 2018.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[19] Emmanuelle Dusserre and Muntsa Padró. Bigger does not mean better! we prefer specificity. In *IWCS 2017 — 12th International Conference on Computational Semantics — Short papers*, 2017.

[20] Douwe Kiela, Felix Hill, and Stephen Clark. Specializing word embeddings for similarity or relatedness. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2044–2048. Association for Computational Linguistics, September 2015.

[21] Kirk Roberts. Assessing the corpus size vs. similarity trade-off for word embeddings in clinical NLP. In *Proceedings of the Clinical Natural Language Processing Workshop (ClinicalNLP)*, pages 54–63. The COLING 2016 Organizing Committee, 2016.

[22] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information, 2017.

Appendix

BERT and Linear Layer Outline

```
          (11): BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (query): Linear(in_features=768, out_features=768, bias=True)
                (key): Linear(in_features=768, out_features=768, bias=True)
                (value): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
              (output): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=True)
            )
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
        (pooler): BertPooler(
          (dense): Linear(in_features=768, out_features=768, bias=True)
          (activation): Tanh()
        )
      )
      (fc): Linear(in_features=768, out_features=18, bias=True)
      (dropout): Dropout(p=0.25, inplace=False)
    )
```

Examples of tickets.

1. 'Goodmorning, I need two VMs with 4 cores and 16GB of RAM each. I also need them to have a 100GB SSD disk. Many thanks'

2. 'Hi, I urgently need for tomorrow morning a 4 CPU server with 32 gigabyte of memory and 1TB of RAID5 disk. I also need 256GB of SSD for fast access of database indexes. Many thanks'

3. 'Hello, can you please allocate a virtual machine with 8 cores/16GB/200GB disk? Thanks'

4. 'Hello, I need 6 machines with 32GB or RAM each. Thanks'

5. 'Is this the right place to ask for a 2 cpu, 16G of RAM, 128 SSD? Many thanks'

6. 'Hello, since the launch of the new project, the cluster is suffering excessive load and therefore we would ask for 4 more servers of the same size. We need plenty CPUs and also plenty memory. Many thanks'

7. 'Hi, can you please increase the disk of server PRIMARY of 100GB?'

8. 'Hello, I need 4 more cores on the main loadbalancer, ALPHA1. The modification can be executed overnight to avoid disrupting operations. Thanks'

9. 'Hi, I think I need 16 more gigabytes of memory for the webserver "gateway" and 4 more CPUs for the application server 3 that sits behind. Let me know where you can schedule the maintenance so that I can pull them out from traffic, thanks.'

Extract of thesaurus words

processor, mainframe, microprocessor, computer, microcontroller, hardware, motherboard, input/output, pentium, instruction, set, minicomputer, arithmetic, logic, unit, peripheral, instruction, simd, software, supercomputer, central, processing, unit, central, processor, computer, hardware, integrated, circuit, processor, register, multicore, processor, pda, computer, science, computing, programmable, computation, processors, mac, macintosh, personal, computer, ibm, risc, z80, mips, ghz, mhz, eniac, memory, transistor, relay, hertz, vector, processor, 8-bit, powerpc, operands, mainframe, computer, system/360, microprogram, floating-point, opteron, pdp-8, harvard, mark, i, mmx, vacuum, tube, microcomputer, c.p.u., bios, operand, compute, emulator, qemu, vmware, workstation, full, virtualization, just-in-time, compilation, java, virtual, machine, virtualbox, cp-67, cpu, kernel, linux, sandbox, cp/cms, interpreter, java, c, fortran, p-code, machine, computer, architectures, operating-system-level, virtualization, popek, and, goldberg, virtualization, requirements, hypervisor, virtual, memory, time-sharing, time, sharing, conversational, monitor, system, memory, overcommitment, compiler, kernel, same-page, merging, microsoft, windows, embedded, system, euler, simmon, pascal, real-time, operating, system, system, platform, high-level, programming, language, vm, parrot, virtual, machine, o-code, .net, framework, common, language, runtime, bcpl, abstraction, layer, infocom, disc, saucer, diskette, floppy, platter, record, floppy, disk, phonograph, record, phonograph, recording, disk,

acetate, disk, audio, circle, round, discus, harrow, plate, round, shape, planchet, magnetic, disc, magnetic, disk, phonograph, recording, cd, memory, device, mp3, hard, disk, discs, desktop, laptop, storage, megabytes, hardware, server, socket, gigabyte, computer, files, drives, motherboard, arrays, autochanger, computing, token, puck, turn, frisbee, diaphragm, plough, husbandry, farming, plow, agriculture, point, dot, lp, disks, seventy-eight, l-p, deadeye, 78, removable, accretion, portable, circuitry, embedded, compressed, stack, compression, rom, plug, sensor, circular, digital, ipod, changer, gigabytes, keyboard.

Extract of Wikipedia articles

Extract of Wikipedia categories and articles to build the embeddings; grand total of articles is 6.311.

Category:Unix , Category:Real-time operating systems,Category:Embedded operating systems ,Category:Operating system families ,Category:Window-based operating systems ,Category:Operating system technology ,Category:Operating system APIs ,Category:Game console operating systems ,Category:Operating systems by architecture ,Category:Software by operating system ,Category:Free software operating systems ,Category:Proprietary operating systems ,Comparison of operating systems ,Category:Time-sharing operating systems ,Category:Educational operating systems ,Category:Lisp (programming language)-based operating systems ,Category:Mobile operating systems ,Category:Just enough operating systems ,Category:Floppy disk-based operating systems ,Category:Discontinued operating systems ,Category:ROM-based operating systems ,Category:Robot operating systems ,Bare machine ,Category:Operating system stubs ,Category:Wikipedia categories named after operating systems ,Glossary of operating systems terms ,Category:Network operating systems ,Category:Operating system distributions bootable from read-only media ,Distributed operating system ,Usage share of operating systems ,Category:Supercomputer operating systems ,Category:Books on operating systems ,Network operating system ,Just enough operating system ,Supercomputer operating system ,History of operating systems ,Category:Operating system criticisms ,Category:Operating system advocacy ,Category:Operating system people ,Real-time operating system ,Category:Microkernel-based operating systems ,Category:Distributed operating systems ,Operating system ,Category:Object-oriented operating systems ,Sugar (software) ,Single address space operating system ,Category:Operating system distributions bootable from external media ,Category:Disk operating systems ,Visopsys ,Memory management (operating systems) ,List of operating systems ,NSDOS ,Zorin OS ,Mobile operating system ,Category:Lists of operating systems ,Linux

kernel ,Friend (operating system) ,Kernel (operating system) ,BridgeOS ,Timeline of operating systems ,Dreamshell ,SymbOS ,Linux ,Category:MSX-DOS ,OSEK ,JavaOS ,OpenBSD security features ,XMK (operating system) ,Java Card OpenPlatform ,OS2000 ,Tandy Video Information System ,BrickOS ,Category:Windows CE ,RTEMS ,Versatile Real-Time Executive ,Matchbox (window manager) ,TinyOS ,Convergent Linux Platform ,DSPnano RTOS ,Rockbox ,Series 80 (software platform) ,Category:Embedded Linux ,NxtOSEK ,Windows IoT ,Comparison of real-time operating systems ,Nano-RK ,Cosmos (operating system) ,IGUANA Computing ,Passport Carrier Release ,Scout (operating system) ,Symobi ,Category:Palm OS ,Category:Windows Mobile ,Category:Windows Embedded Automotive ,Series 30 ,RTXC Quadros ,Smallfoot ,Symbian Foundation ,FreeDOS ,TrueCookPlus ,Category:Lightweight Unix-like systems ,Series 40 ,PikeOS ,ERIKA Enterprise ,ECos ,Plan 9 from Bell Labs ,MQX ,LiteOS ,BeRTOS ,QP (framework) ,LynxOS ,OS-9 ,A/ROSE ,OpenComRTOS ,RT-Thread ,FunkOS ,DioneOS ,Windows Embedded Industry.

Examples of artificially built tickets.

Hi, is it possible to ask for 43 machine with 14 microprocessor 20 of drive, 151 hard disk.

Please let me know as soon as possible. Thanks',

'Could you please allocate for 7 integrated circuit 89 random access memory, 198 storage on 37 host. Thanks.',

'Goodmorning, I need 117 player with 11 processors 107 apples, 71 plate',

'Hi, is it possible to ask for 40 servers with 77 central processor 40 drive, 200 hard disk. Please let me know as soon as possible. Thanks',

'Could you please allocate for 90 processors 127 of battering ram, 88 magnetic disk on 196 client. Thanks.',

'Goodmorning, I need 121 player with 153 processors 44 memory, 113 drives',

'Hi, is it possible to ask for 76 workstation with 71 central processor 32 of battering ram, 200 drives. Please let me know as soon as possible. Thanks'

Results

'Goodmorning, I need two VMs with 4 cores and 16GB of RAM each. I also need them to have a 100GB SSD disk. Many thanks'

'server': [None, None], 'cores': ['4', 'core'], 'ram': ['16', 'ram'], 'disk': ['100', 'disk']

'Hi, I urgently need for tomorrow morning a 4 CPU server with 32 gigabyte of memory and 1TB of RAID5 disk. I also need 256GB of SSD for fast access of database indexes. Many thanks'

'server': [1, 'server'], 'cores': ['4', 'cpu'], 'ram': ['32', 'memory'], 'disk': ['1', 'disk']

'Hello, can you please allocate a virtual machine with 8 cores/16GB/200GB disk? Thanks'

'server': [1, 'machine'], 'cores': [None, 'core'], 'ram': [None, None], 'disk': [None, 'disk']

'Hello, I need 6 machines with 32GB or RAM each. Thanks'

'server': ['6', 'machine'], 'cores': [None, None], 'ram': [None, 'ram'], 'disk': [None, None]

'Is this the right place to ask for a 2 cpu, 16G of RAM, 128 SSD? Many thanks'

'server': [None, None], 'cores': ['2', 'cpu'], 'ram': ['16', 'ram'], 'disk': ['128', 'ssd']

'Hello, for an important project which will be delivered on 24th March, I need a full MySQL database with 150GBs of disk. The server needs to be able to sustain 1000 qps. Thanks'

'server': [1, 'server'], 'cores': [None, None], 'ram': [None, None], 'disk': ['150GBs', 'disk']

'Hello, I need 4 more cores on the main loadbalancer, ALPHA1. The modification can be executed overnight to avoid disrupting operations. Thanks'

'server': [None, None], 'cores': ['4', 'core'], 'ram': [None, None], 'disk': [None, None]

'Hi, I need: 3 servers with: 2 vCPUs; 24GB RAM; 50GB of disk.'

'server': ['3', 'server'], 'cores': [None, None], 'ram': ['24', 'ram'], 'disk': ['50', 'disk']