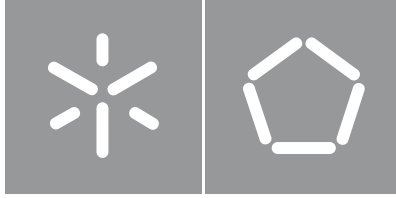




Universidade do Minho
Escola de Engenharia

Daniel Jorge Barros Tinoco

Automated Order Processing and Responses



Universidade do Minho

Escola de Engenharia

Daniel Jorge Barros Tinoco

Automated Order Processing and Responses

Master dissertation

Master Degree in Integrated Master's in Informatics
Engineering

Dissertation supervised by

António Luís Sousa

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorisation conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

LICENSE GRANTED TO USERS OF THIS WORK:



CC BY-NC

<https://creativecommons.org/licenses/by-nc/4.0/>

ACKNOWLEDGMENTS

I would like to thank my friends and family for all the support throughout this dissertation.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

This dissertation was carried out at Uilmédica - Produtos Medicos Hospitalares, Lda which was born on June 9, 2004, as a result of the perception of gaps in the market for the supply of medical - hospital products and equipment to health professionals. One of their main goals is to provide healthcare professionals with the best solutions for the noble mission of ensuring the welfare of us all.

Due to this vision and consequent growth, the company's working methods also need to grow. Therefore, the company wanted to find a solution to the growing quotation requests by the various messaging platforms in which they are present. Starting from this problem, a system was developed that identifies the products contained in a given message and sends an automatic quotation reply.

This system was named Rissa and to develop it, it was necessary to analyse the content of previous **email** messages, in order to develop a **NLP** model that could identify the entities present in future **email** messages. In addition to this, Rissa also contains a search system that filters only the products available from the company.

Rissa had to integrate into an existing infrastructure without impacting the company's operation. This integration had to deal not only with external services, but also with internal services and privacy policies.

In the end, this system was implemented in the company in a real work situation to obtain production results.

Keywords: Email, Natural language processing, Named-entity recognition, Language models, Database text search, Application development, Rissa

RESUMO

Esta dissertação realizou-se na empresa Utilmédica - Produtos Medicos Hospitalares, Lda que nasceu a 9 de Junho de 2004, como resultado da perceção de lacunas no mercado para o fornecimento de produtos e equipamentos médicos - hospitalares aos profissionais de saúde. Um dos seus principais objetivos é fornecer aos profissionais de saúde as melhores soluções para a nobre missão de garantir o bem-estar de todos nós.

Devido a esta visão e consequentemente ao crescimento, os métodos de trabalho da empresa também precisam de crescer. Por isso a empresa gostava de encontrar uma solução para o crescente pedido de orçamentos pelas várias plataformas de mensagens. Partindo deste problema, foi desenvolvido um sistema que identifica os produtos contidos numa determinada mensagem e envia uma resposta automática de orçamento.

Este sistema foi apelidado de Rissa e para o desenvolver foi necessário analisar o conteúdo das mensagens de [email](#) anteriores de modo a desenvolver um modelo de [NLP](#) que fosse capaz de identificar as entidades nas mensagens de [email](#) futuras. Para além disto, Rissa contém um sistema de pesquisa de modo a filtrar apenas os produtos disponibilizados pela empresa.

Rissa teve de se integrar numa infraestrutura já existente sem afetar o funcionamento da empresa. Esta integração teve de lidar não só com os serviços externos, mas também com serviços e políticas de privacidade internas.

No final, este sistema foi implementado na empresa numa situação de trabalho real para se obter resultados de produção.

Palavras-chave: Email, Processamento de linguagem natural, Reconhecimento de entidade mencionada, Modelos de linguagem, Pesquisa de texto na base de dados, Desenvolvimento de aplicação, Rissa

CONTENTS

1	INTRODUCTION	2
1.1	Objectives and goals	2
1.2	Document structure	3
2	STATE OF THE ART	4
2.1	Background	4
2.2	Available solutions	5
2.2.1	Mailparser.io	5
2.2.2	Email Parser by Zapier	5
2.2.3	Email Parser by FrozenFrog Software	6
2.2.4	Parseur	7
2.2.5	Summary and problem division	8
2.3	Email	9
2.3.1	Operation	10
2.3.2	Message format	10
2.4	Natural Language Processing	11
2.4.1	Named-entity recognition/Part-of-speech	11
2.4.2	Traditional	12
2.4.3	Distributed representations	12
2.4.4	Neural networks language models	17
2.4.5	Transformer model	19
2.5	Selected models	24
3	THE PROBLEM AND ITS CHALLENGES	25
3.1	SMTP server	25
3.2	Information extraction	26
3.3	Quote document	27
3.3.1	Database search	27
3.4	Proposed approach - Rissa	27
3.4.1	System architecture	28
3.4.2	Application technologies	30
4	DEVELOPMENT	32
4.1	Dataset	32
4.1.1	Preprocessing	33
4.1.2	Manual entity tagging	34
4.2	Training models	38
4.2.1	Model results	39

4.3	Rissa	39
4.3.1	Product information ingress	40
4.3.2	Inbound SMTP server	42
4.3.3	Information extraction	43
4.3.4	Quote document generation	45
4.3.5	Benchmarks	46
4.4	Summary	47
5	CASE STUDY	48
5.1	Experimental setup	48
5.2	Results	49
5.3	Discussion	50
6	CONCLUSION	51
6.1	Conclusions	51
6.2	Future work	51
	Glossary	59
	Acronyms	60
A	SUPPORT MATERIAL	62

LIST OF FIGURES

Figure 1	Example of extraction features [82].	5
Figure 2	Teach the Parser how to read email [81].	6
Figure 3	Processing email example with the tool Email Parser by FrozenFrog Software [83].	7
Figure 4	Example of data captured by a Parseur template for Zillow [79].	8
Figure 5	Illustration of the participants in an email message exchange [86].	10
Figure 6	Generated markup for named entities from spaCy [75].	11
Figure 7	Generated markup for part-of-speech from spaCy [75].	12
Figure 8	An illustration of distributional vectors of food, eat and laptop [57].	13
Figure 9	Neural Language Model. $C(i)$ is the i^{th} word embedding [66].	13
Figure 10	The Continuous bag-of-word model [43].	14
Figure 11	The skip-gram model [43].	15
Figure 12	Architecture language model applied to an example sentence [51].	16
Figure 13	Convolutional approach to character-level feature extraction [46].	17
Figure 14	Recurrent Neural Network based Language Model [38].	18
Figure 15	Long Short-Term Memory network [84].	19
Figure 16	The Transformer model architecture [60].	20
Figure 17	Overall pre-training and fine-tuning procedures for Bidirectional Encoder Representations from Transformers (BERT) [68].	21
Figure 18	Comparison of a single language modelling (MLM) similar to BERT , and the Cross-lingual Language Model (XLM) dual-language modelling (TLM) [71].	23
Figure 19	Example of the pooled Contextualized embedding generation [67].	24
Figure 20	High level view of the system.	28
Figure 21	System Architecture.	30
Figure 22	Exchanged emails in a conversation and it's occurrence.	33
Figure 23	Illustration representing the tagging-training-correct flow.	36
Figure 24	Occurrence of Named-entity recognition (NER) tags.	37
Figure 25	Run time of the three main parts of the function present in listing 9.	47
Figure 26	Model memory usage per second, during initialization time.	48
Figure 27	Processed email messages per minute.	49

LIST OF TABLES

Table 1	Comparison between available solutions	9
Table 2	The mean F_1 score of every model for each chosen label.	39
Table 3	Exchanged emails in a conversation and it's corresponding occurrence accompanied with the occurrence percent value.	62

LIST OF LISTINGS

1	Substitute all occurrences of one or more whitespaces for a common space.	34
2	Example of DistilBERT configuration file.	38
3	Example of the body Hypertext Transfer Protocol (HTTP) request to Rissa.	40
4	Representation of the <i>products</i> schema using Ecto.	41
5	Information indexed for each product in Elasticsearch.	42
6	Implementation of the <code>handle_DATA/4</code> callback from the behaviour <code>gen_smtp_server_session</code> .	42
7	Excerpt from the Python file that extract entities.	43
8	Function <code>init/1</code> for the <code>Rissa.Ner.GenServer</code> .	44
9	Email message flow from after being received to before being sent.	45
10	Quote email response construction function with the option to also add a Portable Document Format (PDF) file.	46
11	Convert the integer product price to a human friendly string.	46
12	Elasticsearch query example for text <i>Máscaras MarcaA</i> .	63

INTRODUCTION

The need to automate repetitive tasks in businesses has always been essential on many levels. For this reason, studying and understanding the tools and technologies available, in particular, the request for quotations/budgets by textual ways has been increasing due to the emergence and promotion of multiple messaging platforms. There is thus the need to follow this growth more automatically.

In this sense, this dissertation was carried out in a business context, where a company presented the problem of the possibility of the automation of the process of responding to requests for quotes, since such automation represents gains for the company, freeing up time for employees to perform other tasks and possibly increasing the conversion of sales, due to a faster responses.

This problem, at first sight, seems simple because it involves usual communication methods, however the approach to the problem will focus on the textual content of these messages, therefore it will be necessary to recourse to [NLP](#) technology.

Every day, the company receives several quotation requests. These requests come from different platforms, namely the sales team emails, the website contact form, and instant messaging platforms. Each employee handles this request at their discretion. This process is many times the same because some clients order almost the same products and even give out the reference of the products they want to buy, while others order the products by name, many times not even using the correct name for them.

1.1 OBJECTIVES AND GOALS

The main goal of this dissertation is to solve this problem by carefully analysing it and formulate a solution. With that in mind, we will start by analysing several messages and all the platforms involved. Analyse the content of the messages and try to automate the process of response by finding the right technology for this case.

This dissertation will research for state of the art products to see if none solve the main problem at hand, and having none found will focus on developing a system that can respond automatically to the quotation requests.

By the end of this work, a solution that solves the company's problem will be presented and deployed in the real world, in order to ascertain it's precision in the generated response.

1.2 DOCUMENT STRUCTURE

Besides this introductory chapter, this dissertation is organized in the following chapters:

- Chapter 2 presents possible market solutions and the main research made into the main problem area.
- Chapter 3 describes the main problem parts it's challenges, along with the system's architecture.
- Chapter 4 displays the steps taken to develop the system proposed in Chapter 4.
- Chapter 5 presents the results obtained from using the system in a real world scenario.
- Chapter 6 ends this dissertation by reflecting on the overall work and future improvements.

STATE OF THE ART

Written messages are a part of everyday life for businesses and individuals alike. This Chapter will begin by discussing the context in which the need for this project appears and give an overview about the fundamental features in it. We will also research similar solutions, present them and explain what they miss to be a viable solution to this problem.

As no immediate market solution presents itself as having all the minimum requirements for the project, this chapter will also discuss some state of the art solutions and technologies for each of the main parts of the project.

2.1 BACKGROUND

As a company grows, so do its needs to automate specific tasks of its daily routine. In this particular case, many costumers send quoting requests for several catalogue products. These requests for quotes come from several sources, but mainly in the form of formal **email**.

The employees in charge of handling the incoming **emails** need to read them and proceed to understand what is being referred by the sender. This process can be an easy task since many costumers already send the products they want, referenced with the name or product number. After that, they need to look up the products described and prepare a quotation to send back to the client. Following the quotation confirmation, it is then sent back as a reply to the customer, and a new internal process begins.

This project focus will be in the incoming **email** requests, their automated processing and quotation generation. Also, since the time this process takes is an essential factor, the quotation generation phase may also be automated as to decrease the overall time to send back a response.

An essential factor to be taken into account for the project is that the majority of the **email** content is written in **Portuguese** being the language that will have a greater focus.

Having the overall project described, it is necessary to find a solution that can handle the incoming **email** requests and parse them. A process that involves finding the meaning of the message content and then search for it in an internal database, drafting a response, confirm it and send it back to the customer.

2.2 AVAILABLE SOLUTIONS

There are many market solutions throughout the [World Wide Web \(WWW\)](#); in this section, we will give an overview of some of them, presenting their current abilities and limitations.

2.2.1 Mailparser.io

Mailparser [82] is a [Software as a Service \(SaaS\)](#) company that provides a product capable of parsing incoming [email](#) and in return, respond with structured data.

The creation of Mailparser follows the discovery, by the company's founder Moritz Dausinger, of the global need for automating the workflow of inbound [emails](#). In the following 36 months of operation Mailparser grew quickly leading to several acquisition offers, and by 2017 Mailparser was acquired by SureSwift Capital.

This product allows the creation of custom [email](#) parser rules, giving the ability to process and tailor the data as needed. A rule is typically a series of steps where it manipulates the [email](#) content and filters out the content not needed until only the crucial data remains.

Another feature is their integrations with several other services and export capabilities, allowing them to move the resulting data with ease.

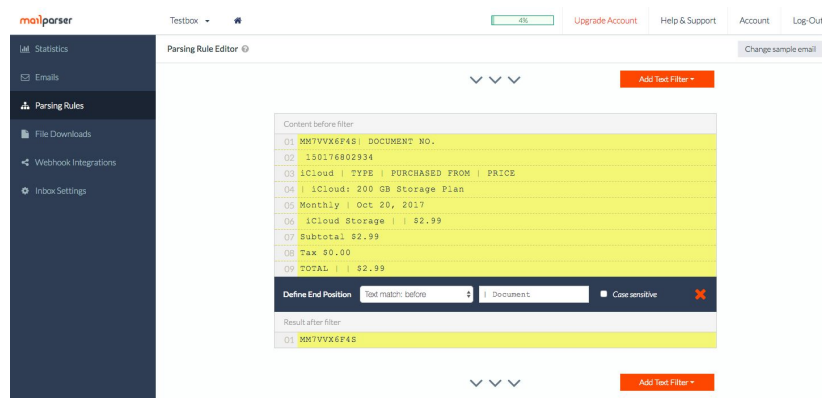


Figure 1: Example of extraction features [82].

This service has many pricing tiers, and from their pricing page, it is possible to conclude that the prices vary from 39 \$ to 299 \$ per month with the possibility of contact for more significant [email](#) handling volume.

2.2.2 Email Parser by Zapier

Zapier's Email Parser [81] is a tool to copy text out of emails to pass the extracted text to another tool allowing for better integration between different tools.

Email parsing and text extraction with Zapier is only the first step as Zapier has over 1500 integrations with other services, making it one of the platforms found with the higher number of integrations.

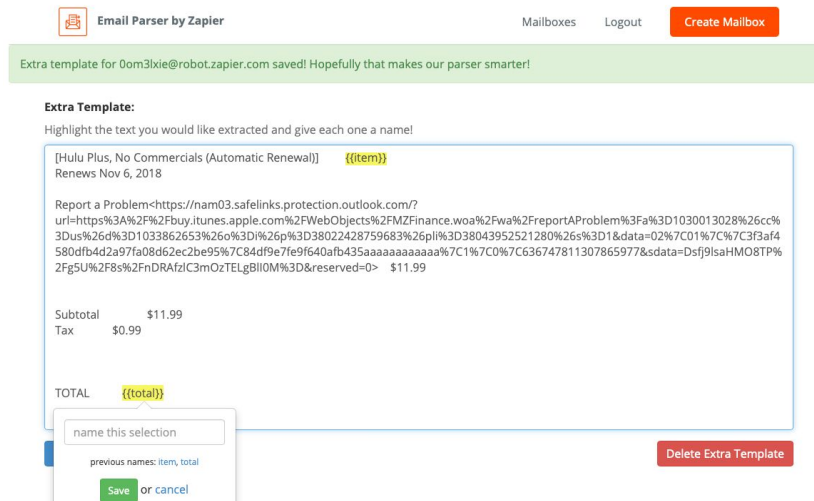


Figure 2: Teach the Parser how to read email [81].

This service is easy to use, and an employee could be tasked with setting it up, as shown in Figure 2. However, it is only designed to parse text from emails, and may fail at copying text if the email formatting changes frequently, as usually is the case with user-generated messages.

This tool from Zapier is free to use, and they also offer a free plan (0 \$), however, from their pricing page their prices go up to 599 \$ per month when billed annually. Since Zapier does not only handle email, their tiers are measured in the number of tasks per month and not email volume.

2.2.3 Email Parser by FrozenFrog Software

Email Parser [83] is a tool developed by FrozenFrog Software with the purpose of extracting data from incoming emails, thus automating the user's workflow.

As with other services, the primary purpose of this tool is to handle incoming emails by parsing them with a set of pre-defined rules by the user. The rules can be simple, as simple filters, filtering by address or message content, or they can be more complex using script actions in which an user can run a program with the help from their Software development kit (SDK) being limited to the programming language C# [21].

Besides parsing email, it is also possible to configure this tool to send automated responses defined by user email templates.

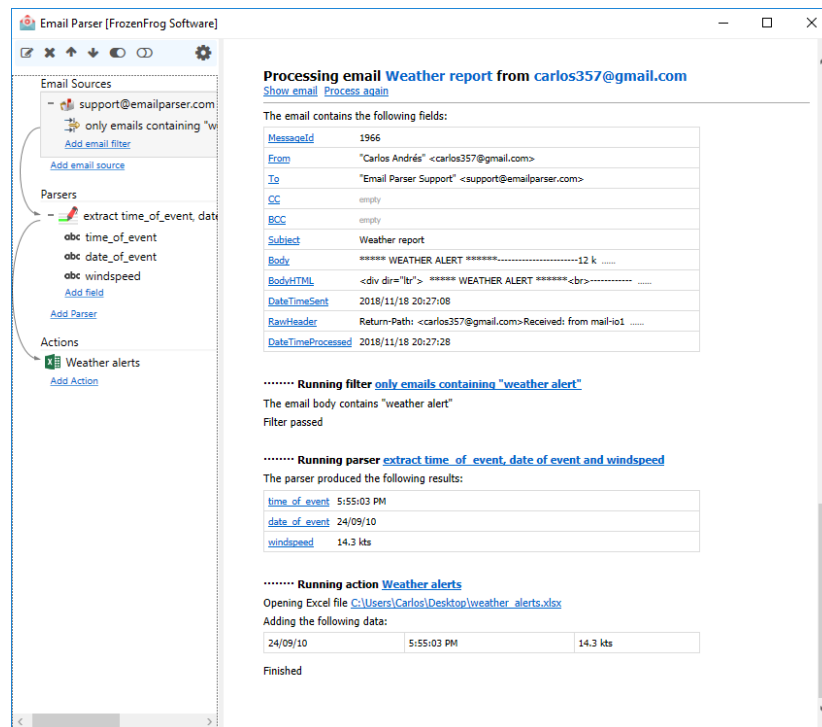


Figure 3: Processing email example with the tool Email Parser by FrozenFrog Software [83].

The Email Parser by FrozenFrog Software has two pricing modes, a license key as a one-time payment with one year of free updates and an unlimited number of emails. In this option, emails are processed on-premises, meaning in the user's computer. Another pricing option is Email Parser Online in which the user's configurations are uploaded to the cloud, and the software is then run there without the need for dedicated computer infrastructure on-premises. This service has the cost of 79.95\$, one-time payment, or 9\$ per month with the option to cancel at any time.

2.2.4 Parseur

Parseur [79] is an email parsing solution, that aims at keeping a simple point and click approach. Parseur main purpose is to be used to create a workflow and automate text extraction from emails and then send the parsed data to applications via their integrations.

Besides email, Parseur can handle the parsing of attachments, documents, spreadsheets, and PDFs.

MessageType

You have a new tour request regarding 456 Other Road, Saint Louis, MO, 63135.

CustomerName requested a tour:

Availability

" **Message** "

Reply directly to this email or


[SMS text](#)

[Call](#)

[See John's phone contact info](#)

Cristalle found your listing on: [Zillow](#)

Manage your listing



PropertyPrice /mo

PropertyDetails

PropertyAddress

TransactionType

Figure 4: Example of data captured by a Parseur template for Zillow [79].

This service has many pricing tiers, with several benefits and credits to use in the platform. Each document process spends one credit. From their pricing page, it is possible to conclude that the prices vary from 0 €(very limited) to 249 €per month with the possibility of contact for more significant document handling volume.

2.2.5 Summary and problem division

From the research above, it is possible to conclude that this kind of services are generally more focused for the office employee with their emphasis on ease of use and their integrations with several other business tools.

The focus is also on the actively parsing of well-defined [email](#) templates, as they all need a set of rules to be pre-defined when searching for important data, making it difficult to extract information on unstructured [emails](#).

In order to cater to all of the requirements of this project, these services would need much time invested in research of their capabilities even further and in integrating the needed business tools with them.

	Mailparser.io	Email Parser by Zapier	Email Parser by FrozenFrog Software	Parseur	Custom Solution
Setup	Rule based	Template based Point & Click	Rule based	Template based Point & Click	Automatic
Post processing	Yes	via multi-step Zap	Yes	Yes	Yes
Integrations	Yes	Yes (Zapier)	Yes	Yes	Yes (custom)
Retention Policy	Yes	No	Yes (on premises)	Yes	Yes (on premises)
Internal database search	No	No	No	No	Yes
Email draft generation	No	No	Yes (send to drafts email folder)	No	Yes
Automatic Response	No	Yes (with integrations)	Yes	No	Yes
Support	Yes	Online documentation	Yes (39.95 \$ per hour)	Yes	Yes (on premises)
GDPR Compliance	Yes	Unknown	Unknown	Yes	Yes (company police)

Table 1: Comparison between available solutions

For the above reasons represented in Table 1, it was concluded that a custom approach would be the wisest and the strategy to follow.

During this research and the assessment of business requirements with the company, it was possible to divide the problem in four high level parts:

- Process the incoming [emails](#);
- Extract the relevant information from each [email](#);
- Create the quote document;
- Send the quote document.

Having the relevant parts of the main problem, the research proceeds into each one.

2.3 EMAIL

As said earlier, the type of incoming messages to handle will be [email](#) messages; as such, a greater understanding of this method of exchanging messages is needed.

[Email](#) is a method that permits to write, send and receive messages through electronic communication systems, in other words, it is a method of exchanging messages between people using electronic devices.

The invention of this communication method is credited to Ray Tomlinson [85] in 1972. Early [email](#) systems required the sender and the recipient to both be online at the same time. However, modern [email](#) systems are based on a store-and-forward model, having the benefit of neither the users nor their computers to be online simultaneously as they now only need to connect briefly to a mail server or a webmail interface for as long as it takes to send or receive messages or to download it.

2.3.1 Operation

Email can be used in several ways, however a typical sequence of events when a sender transmits a message using a Mail User Agent (MUA) addressed to the email address of one recipient, can be illustrated by the following Figure 5.

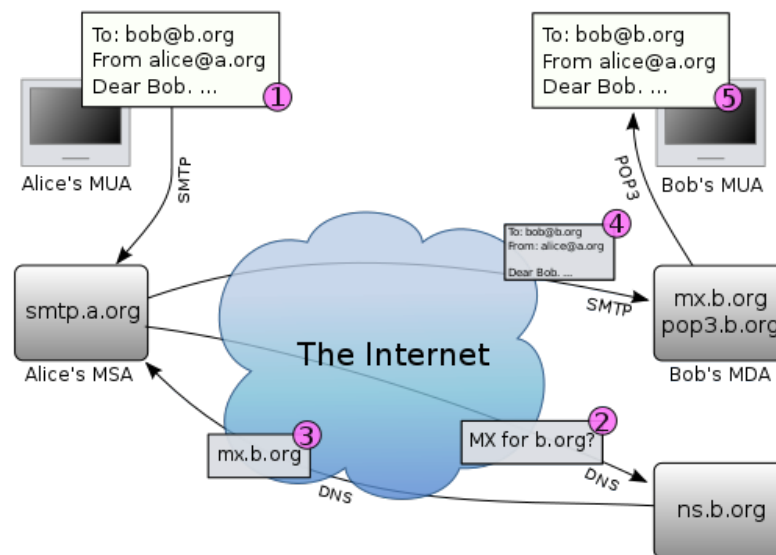


Figure 5: Illustration of the participants in an email message exchange [86].

2.3.2 Message format

The basic Internet message format used for email, is, at the time of this writing, defined by Request for Comments (RFC) 5322 [36], with encoding of non-American Standard Code for Information Interchange (ASCII) data and multimedia content attachments being defined in RFC 2045 through RFC 2049 [16, 17, 13, 14, 15], collectively called Multipurpose Internet Mail Extensions (MIME).

There are several message header fields, but for this use case we will only focus on the From and Subject fields.

In this use case, the message body and its content will be the most crucial part of the entire message.

2.4 NATURAL LANGUAGE PROCESSING

Natural Language Processing is a theoretically motivated range of computational techniques for analysing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human like language processing for a range of tasks or applications. [80]

—Elizabeth D. Liddy

While the full lineage of **Natural Language Processing (NLP)** does depend on several other disciplines, **NLP** strives for human-like performance, it is then appropriate to consider it an **Artificial Intelligence (AI)** discipline [80].

For a range of tasks or applications **NLP** is not usually considered a goal in and of itself, except perhaps for **AI** researchers. For others, **NLP** is the means for accomplishing a particular task or process, being particular examples of this **Information Retrieval (IR)** systems that utilize **NLP**, **Machine Translation (MT)** and many others [80].

2.4.1 *Named-entity recognition/Part-of-speech*

For our particular case, as with many **NLP** problems, we have two subtasks that we will focus the most for this section of work, being **NER** and **Part-of-speech (POS)**.

The **NER** is used to identify different entities in unstructured text and categorize them into pre-defined categories. Normally the **NER** subtask is divided in two distinct problems, the detection of names and their classification into their respectively category [26]. Figure 6 shows the identified entities and their respectively categories using the tool **spaCy** [75].

As exhibited in the name, **POS** is the process of classifying words into their parts of speech and labelling them accordingly. An example of the finalized process is represented in Figure 7.

These two subtasks are key elements in both information extraction systems and in syntactical analysis, respectively [41].

When Sebastian Thrun PERSON started working on self-driving cars at Google ORG in 2007 DATE, few people outside of the company took him seriously.

Figure 6: Generated markup for named entities from **spaCy** [75].

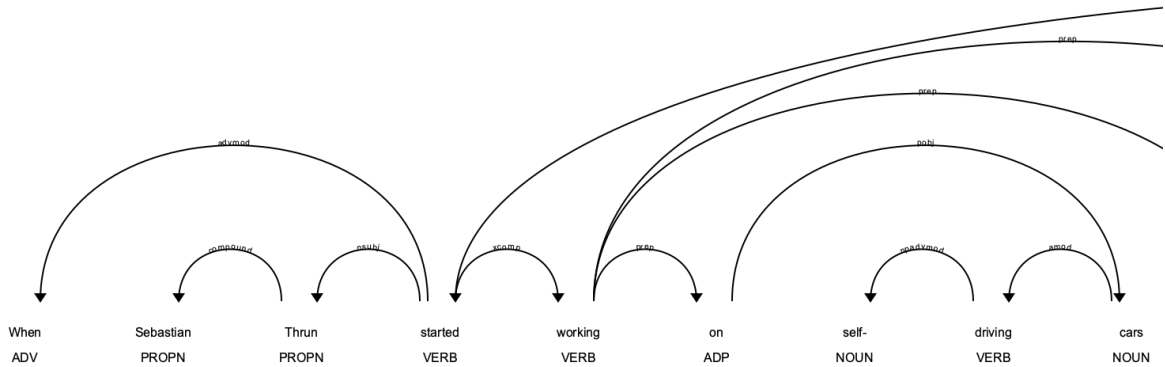


Figure 7: Generated markup for part-of-speech from spaCy [75].

2.4.2 Traditional

During many years, the majority of methods used to study NLP problems employed shallow machine learning models and time-consuming, hand-crafted features [61]. However, this leads to problems such as the *curse of dimensionality*, while learning joint probability functions of language models, since linguistic information was represented with sparse representations (high-dimensional features).

This problem was the motivation behind distributed representations of words existing in low-dimensional space [25] compared to traditional machine learning models like **Support-vector machine (SVM)** [9] or logistic regression [23].

2.4.3 Distributed representations

One important characteristic of a word is the company it keeps. According to the distributional hypothesis [2], words that occur in similar contexts (with the same neighbouring words), tend to possess similar meanings.

These vectors try to capture the characteristics of the neighbours of a word. The main advantage of distributional vectors is that they capture the similarity between words as illustrated by Figure 8.

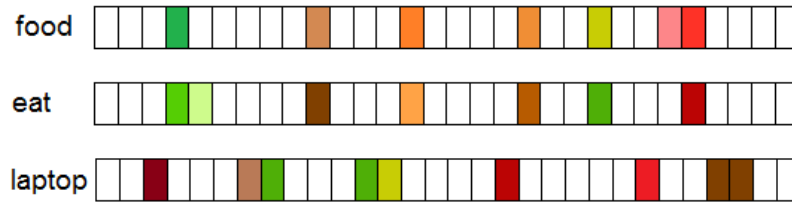


Figure 8: An illustration of distributional vectors of food, eat and laptop [57].

Word embeddings

Typically, word embeddings are pre-trained on a task where the objective is to predict a word based on its context. The basic idea is to store the same contextual information in a low-dimensional vector, but now each word is represented by a N-dimensional vector, where N is a relatively small number (typically between 50 and 1000).

This type of approach was first presented in 2003 by Bengio et al. [25], and can be seen in Figure 9. In their approach, the authors proposed a neural language model which learned distributed representations for words.

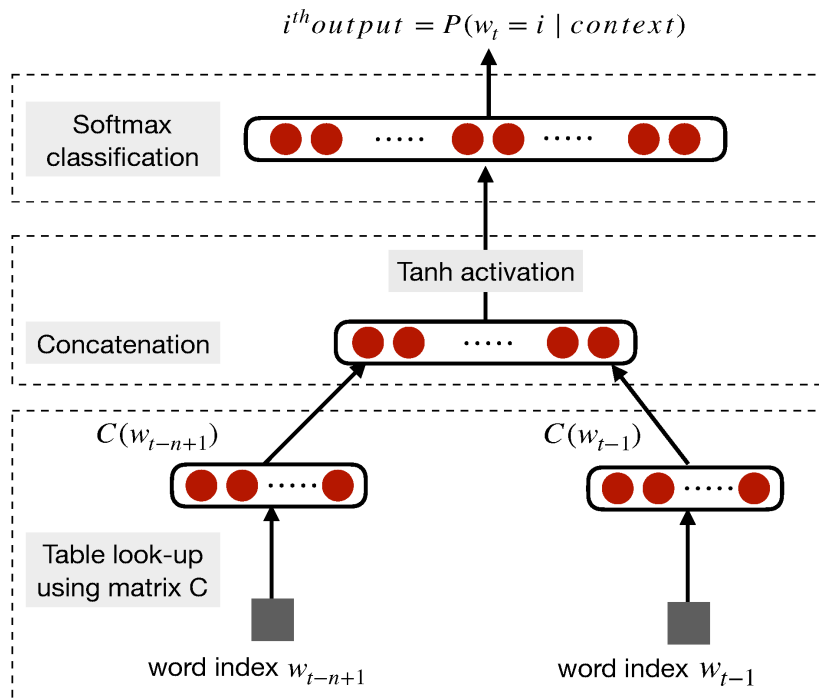


Figure 9: Neural Language Model. $C(i)$ is the i^{th} word embedding [66].

Word embeddings are also able to capture syntactic and semantic information, a feature by itself useful; however, this approach is also useful for tasks such as POS tagging and NER, intra-word morphological and shape information retrieval [46].

Word2vec

In 2013, word embeddings were revolutionized by Mikolov et al. [43] who proposed both the **Continuous Bag of Words (CBOW)** and skip-gram models being considered the pioneer of word embeddings in mainstream deep learning [69].

On the one hand **CBOW** (Figure 10) is a neural approach to construct word embeddings, and the objective is to compute the conditional probability of a target word given the context words in a given window size.

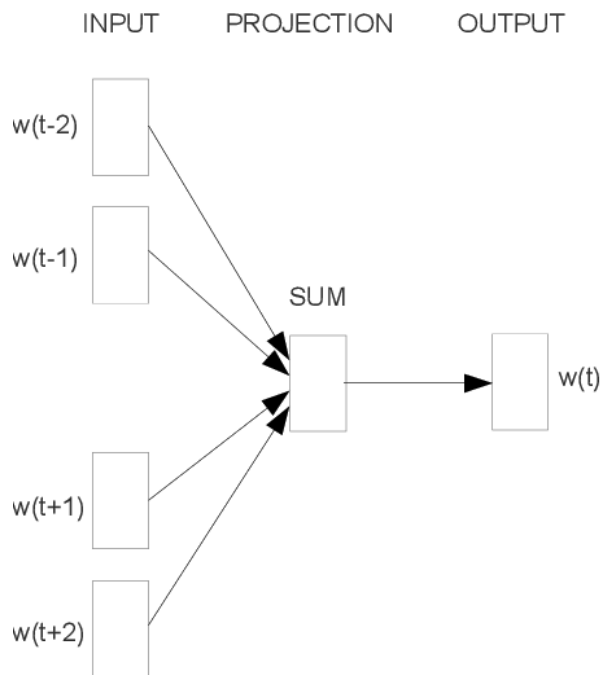


Figure 10: The Continuous bag-of-word model [43].

On the other hand, the skip-gram model (Figure 11) does the opposite of the **CBOW** model by predicting the surrounding context words given the central target word.

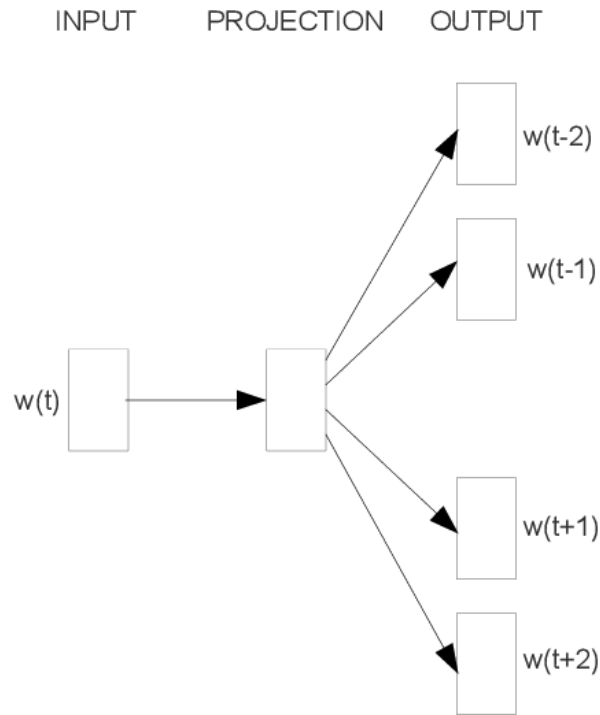


Figure 11: The skip-gram model [43].

For both of these models, in an unsupervised setting, the word embedding dimension is determined by the accuracy of prediction. As the embedding dimension increases, the accuracy of prediction also increases until it converges at some point in time, which then is considered the optimal embedding dimension as it is the shortest dimension without compromising accuracy.

Having in common the use of a single word, both these models have one limitation for individual word embeddings which translates in their inability to represent phrases, where the combination of two or more words do not represent the combination of meanings of individual words [42].

GloVe

Global Vectors for Word Representation (GloVe) is another commonly used method of obtaining pre-trained embeddings using an unsupervised learning algorithm for obtaining vector representations for words [47].

As with Word2vec 2.4.3, **GloVe** relies on interpretability of the embedding vectors and the frequency of co-occurrence of words [48]. As such an association between Word2vec 2.4.3 and **GloVe** is often formed, however they have important differences.

For **GloVe**, the frequency of co-occurrence of words is a piece of central information that guides the learning. Also, instead of using skip-gram or **CBoW** models, **GloVe** minimizes the difference between the product of word embeddings and the log of the probability of co-occurrence using **Stochastic Gradient Descent (SGD)** [48].

It is possible to summarize GloVe as a count-based word embedding approach that learns an optimized, lower-dimensional version of a co-occurrence matrix.

Character embeddings

As previously stated, word embeddings are able to capture morphological information. However, it is possible to also analyse the individual characters in order to have more robust methods of extracting morphological information from words and select which features are most important for the task at hand.

Building natural language understanding systems at the character level has attracted researchers attention, such as the work provided by Kim, et all [51] where they introduce a neural language model that uses only character-level inputs, as represented in Figure 12. This model outperforms baseline models that utilize word embeddings in the input layer while utilizing fewer parameters.

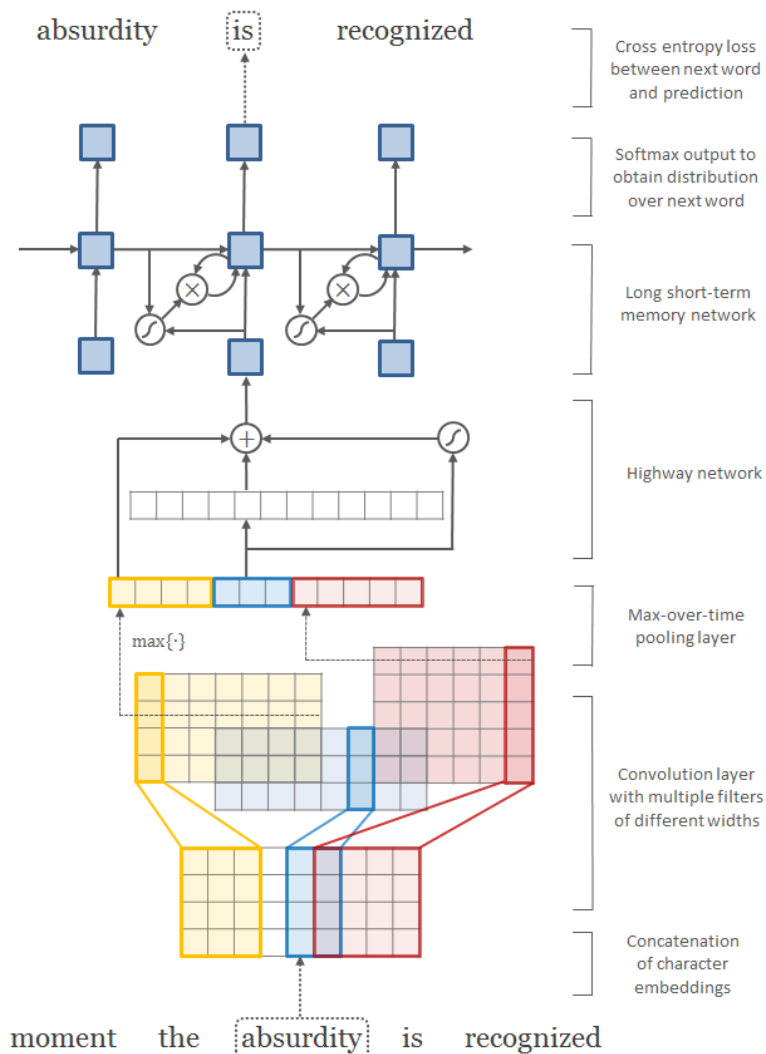


Figure 12: Architecture language model applied to an example sentence [51].

On more morphologically rich languages, such as Portuguese, better results are reported in certain NLP tasks. Santos and Guimarães [52] applied character-level representations, along with word embeddings for NER, achieving state-of-the-art results in Portuguese and Spanish corpora. This model was based on the CharWNN neural network [46](Figure 13) that produces local features around each character of the word and then combines them using a max operation to create a fixed-sized character-level embedding of the word.

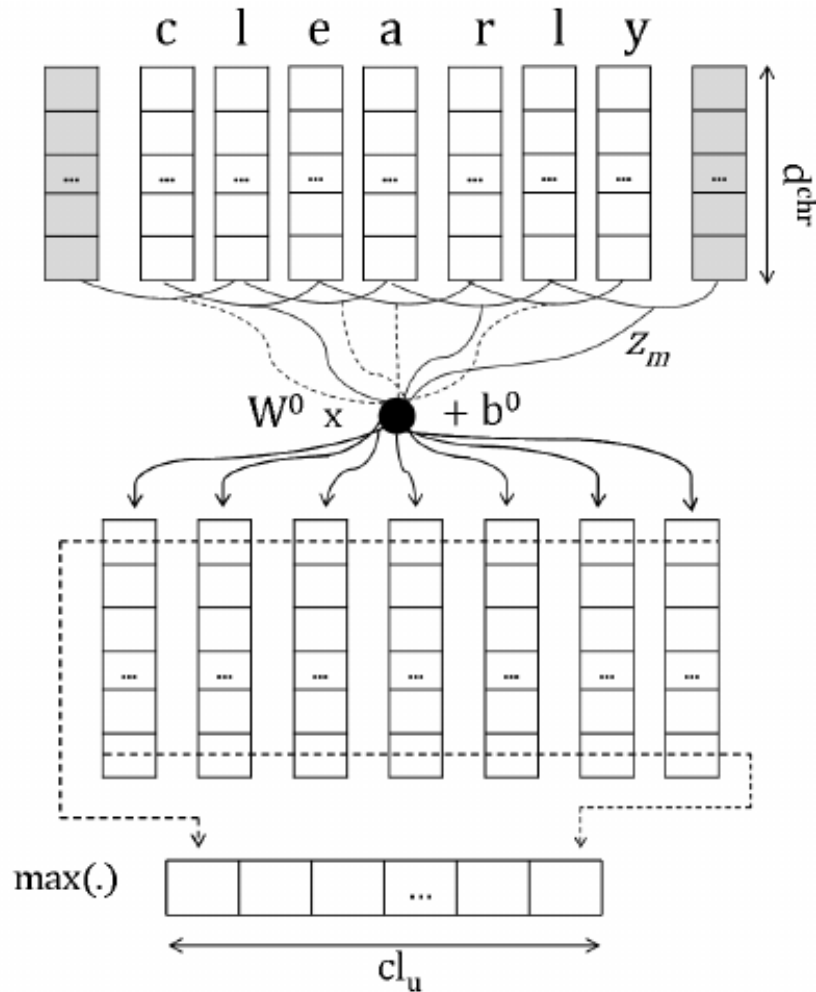


Figure 13: Convolutional approach to character-level feature extraction [46].

2.4.4 Neural networks language models

Besides feed-forward neural networks and the related Word Embeddings(2.4.3), other topologies and architectures of Artificial Neural Networks (ANNs) [6] where tried.

One of those other networks were Recurrent Neural Networks (RNNs), introduced in 1990, RNNs [7] form a directed graph along a temporal sequence, allowing previous outputs to be used as inputs while having hidden states.

For language modelling, specifically for speech recognition, in 2010 a new RNN (Figure 14) language model was introduced having at the time improved upon state of the art results by 18% in reduction of word error rate on the Wall Street Journal task, although at the cost of much higher computational complexity when training [38].

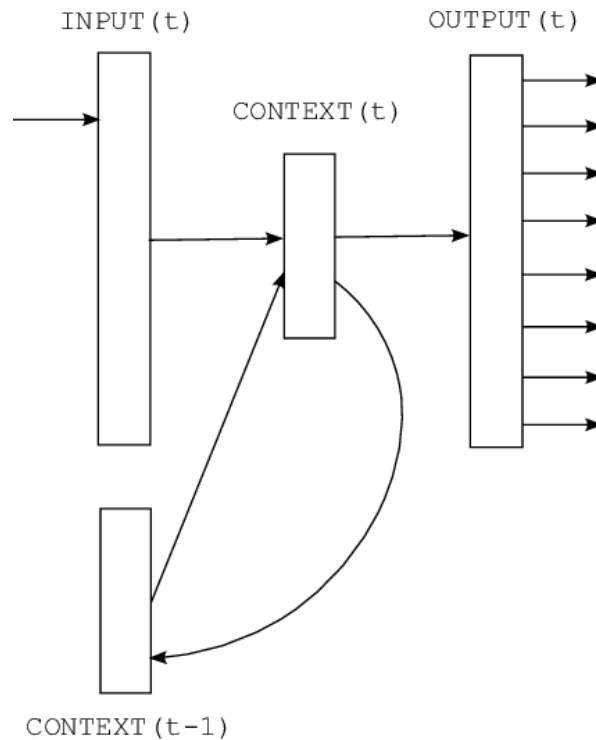


Figure 14: Recurrent Neural Network based Language Model [38].

RNNs with long dependencies were still thought to be difficult to train, until 2013, where Sutskever [44] presented a new variant of the Hessian-free (HF) optimizer for tasks that have long-range temporal dependencies and also a scheme to describe the random parameter initialization, for tasks that have long-term dependencies.

Traditional RNNs [7] have some problems as high computational complexity and also suffer from both the vanishing and exploding gradient problem. The vanishing and exploding gradient problem happen because it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing, for the vanishing problem or increasing, for the exploding problem, with respect to the number of layers. In some cases, the vanishingly small gradient, makes the network take a long time to train or may completely stop the neural network from further training. On other cases the exploding weight recurring value make the networks value's oscillate without learning any meaningful representation. For this reason RNN were replaced by Long Short-Term Memory network (LSTM) [18] which proved more resilient to the vanishing and exploding gradient problem.

LSTM (Figure 15) share the same architecture as RNN, however as a way to solve the vanishing gradient problem the classical LSTM introduced the Constant Error Carousel (CEC).

The classical LSTM block was composed of a cell, an input gate, an output gate [18] and a forget gate, that was later introduced in 1999 [19].

While the introduction of CEC solved the vanishing gradient problem, LSTM did not solve the exploding gradient problem for the RNN architecture [49].

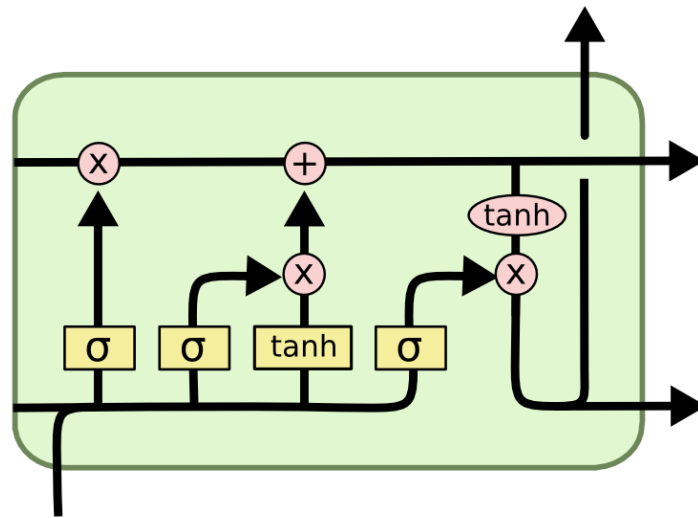


Figure 15: Long Short-Term Memory network [84].

2.4.5 Transformer model

Within the year 2017, NLP saw the beginning of a new type of models with the publication of *Attention Is All You Need* [60]. In this work, a specific type of attention based network is introduced, called the Transformer.

The mechanism of attention in a model was not a new idea, having been used by Bahdanau et al. [54] where the authors propose an encoder–decoder architecture with the caveat of instead of using a fixed-length vector, the model automatically (soft-)searches for the relevant parts to predict the target word. The proposed model uses attention in the decoder part to alleviate the encoder from having to encode a fixed-length vector with all the source information, effectively spreading the information throughout the sequence of annotations, that are later selected by the decoder.

Similar to RNN(2.4.4), the Transformer model (Figure 16) handles it's data in sequence, however no particular order is necessary when inputting it. With this capability, the Transformer model can train faster and, as such, with more data using parallelization.

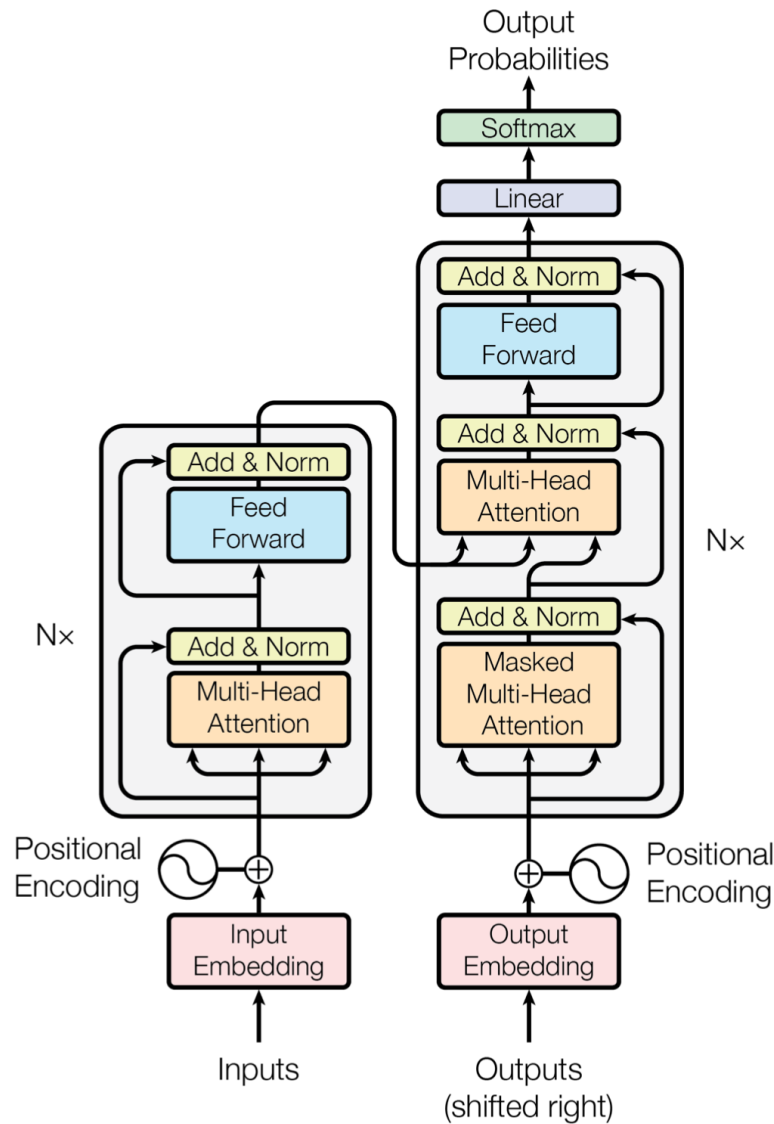


Figure 16: The Transformer model architecture [60].

Although classic pre-trained word embeddings (2.4.3) have been shown to be of great use for downstream NLP tasks [48, 42], the Transformer model led to our current state in NLP.

In the sections underneath, we will show some models that are descendants of the Transformer model and have a **Portuguese** or a multilingual version.

Bidirectional Encoder Representations from Transformers (BERT)

Introduced in 2018, BERT [68] was designed to pre-train deep bidirectional representations from unlabelled text by jointly conditioning on both left and right context in all layers.

BERT is conceptually similar to Word2Vec and GloVe, however BERT instead forms different word vectors for different contexts of the same word.

Previous models, such as ELMo [64] that uses a feature-based strategy, a technique that includes the pre-trained representations as additional features, or OpenAI GPT [65] which uses a fine-tuning based approach, that can be trained on downstream tasks by simply fine-tuning all pre-trained parameters, both share the same objective function during pre-training. These models also use unidirectional language models to learn general language representations which limits the choice of architectures used during pre-training and limits the pre-trained representations. BERT improves the fine-tuning based approach by alleviating the unidirectionality constraint by using a Masked Language Model (MLM) [1] pre-training objective. The MLM works by randomly masking some of the input tokens, and the model objective is to predict the original word from the context only.

BERT argues that bidirectionality is crucial in neural networks as it allows the information to flow forwards and backwards as the model trains, which leads to better model performance.

This makes BERT a deeply bidirectional unsupervised model that can be fine-tuned with just one additional output layer, of labelled data, to create models for several downstream NLP tasks (Figure 17), including for our case, NER.

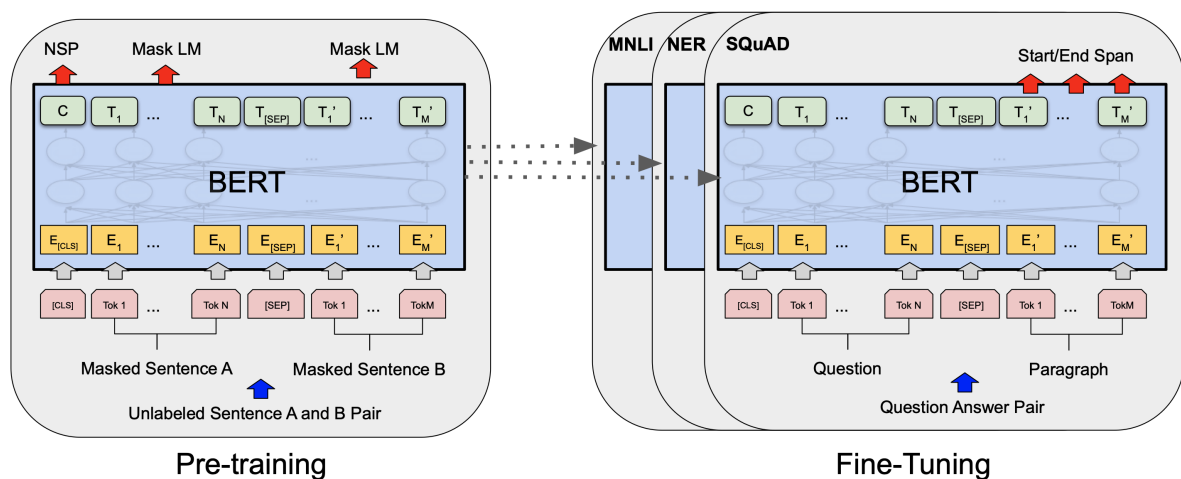


Figure 17: Overall pre-training and fine-tuning procedures for BERT [68].

DistilBERT

DistilBERT [76] is a distilled version of BERT with the purpose of exploring and creating smaller models that are easier to use in more compute constrained environments.

First introduced by Bucila et al. [29] and generalized by Hinton et al. [50], DistilBERT explores Model Compression and Knowledge Distillation of a Neural Network, respectively. This distillation is used to compress a larger model, the teacher, into a smaller model model, the student. The smaller model, the student, is trained to reproduce the behaviour of the larger model, the teacher. Because of this analogy, knowledge distillation is also referred as the teacher-student learning.

The presented DistilBERT, has the same architecture as BERT, however the token-type embeddings and the pooler were removed and the number of layers cut by half. With a smaller model, a reduction of 40%, it was expected that DistilBERT was less performing than BERT, but with more than 95% of the performance retained on the [General Language Understanding Evaluation \(GLUE\)](#) benchmark [72] and 60% faster when inferring.

When comparing with BERT, the smaller size, almost equal performance and faster inference of DistilBERT may represent, for our case, an opportunity to reduce costs and still maintain acceptable results.

XLM

Although BERT has a multilingual model that was trained on 104 languages it wasn't optimized for multilingual models. Most of the vocabulary used to train the multilingual version wasn't shared between languages and therefore the shared knowledge was limited. As such the monolingual versions for English and Chinese are more likely to perform better for fine-tuning in downstream tasks.

XLM [71] extend the generative pretraining for English natural language understanding approach, used by BERT and others, to multiple languages and shows the effectiveness of cross-lingual pretraining.

In order to overcome the limitation of the shared vocabulary between languages, XLM instead of using word or characters as the input of the model, it uses [Byte Pair Encoding \(BPE\)](#) [56], to split the input into the most common sub-words across all languages, thus increasing the shared vocabulary between languages.

BPE can be described as a data compression technique that iteratively replaces the most frequent pair of symbols with a single unused symbol. BPE's algorithm in each input finds and merges the most frequent pair of symbols to create a new symbol. Before the next iteration, the new symbol replaces all the occurrences of the selected pair. The algorithm runs for a predetermined number of iterations where the most that a sequence of symbols can be merged is up to a word.

XLM samples sentences according to a multinomial distribution to reduce bias towards high-resource languages while preventing words of low-resource languages from being split at the character level by increasing the number of tokens associated with them.

XLM also uses the MLM objective and introduces the [Translation Language Modeling \(TLM\)](#) [71] for improving cross-lingual pretraining. The MLM objective is similar to the approach taken in BERT, but instead of pairs of sentences, XLM uses text streams of an arbitrary number of sentences, while also sub-sampling the frequent outputs according to a multinomial distribution, whose weights are proportional to the square root of their inverted frequencies. The TLM objective is in itself an extension of MLM, where rather than using monolingual text streams, XLM concatenate parallel sentences, randomly masking words in both the source and target sentences. This allows XLM to use the context from one language

to predict the masked words in the other language, as shown for English and French in Figure 18.

Both the MLM and TLM objectives are illustrated in Figure 18.

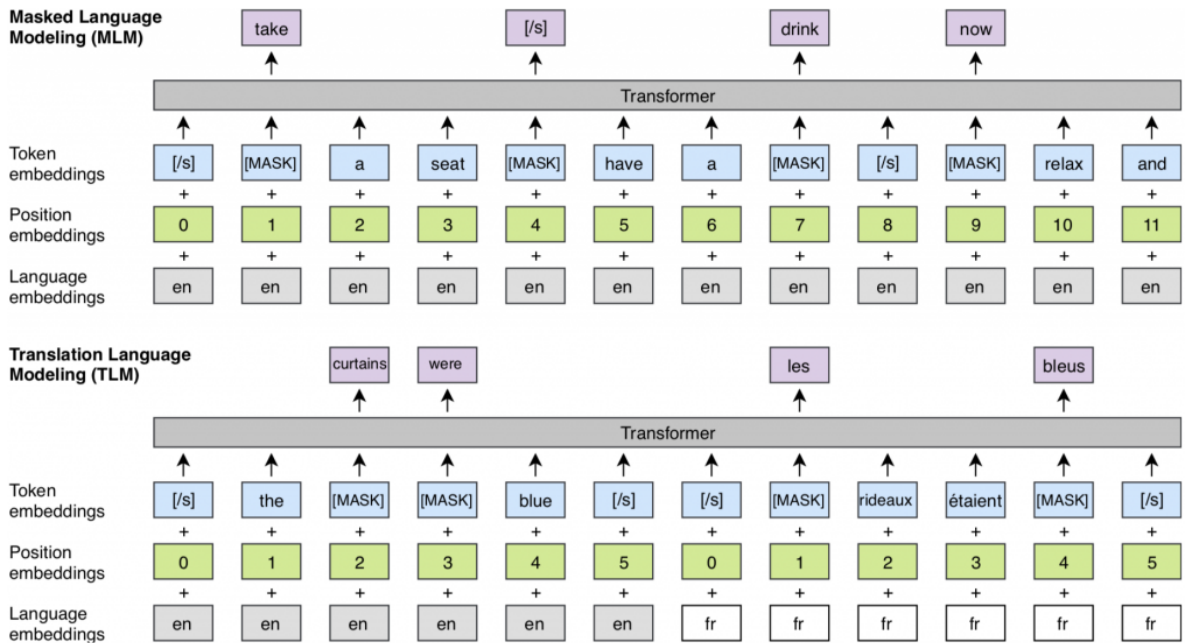


Figure 18: Comparison of a single language modelling (MLM) similar to BERT, and the XLM dual-language modelling (TLM) [71].

XLM-R

XLM-R [73], a cross-lingual language model that closely follows the *XLM* [71] approach, only having minor changes to improve performance at a larger scale. One of this changes is the removal of language embeddings, allowing the model to better deal with code switching.

In *XLM-R*, the Transformer model used is also trained with the multilingual MLM objective where streams of text are sampled for each language the model trained to predict the masked tokens in the input.

Instead of using the same corpus to train as *XLM*, *XLM-R* builds a new corpus from CommonCrawl data in 100 different languages.

For the downstream task of NER, *XLM-R* slightly outperforms the multilingual version of BERT albeit at the cost of a bigger model size and more training time.

Pooled Contextualized Embeddings

Not being a direct descendant of the Transformer model, Pooled Contextualized Embeddings [67] differs in the approach by using character embeddings as opposed to word embeddings.

The approach presented (Figure 19) addresses the common issue with **character embeddings**, where a rare word, one that hasn't appeared in the corpus used to generate the word embeddings, can be misclassified in downstream tasks such as **NER**. To resolve the issue, the model dynamically aggregates contextualized embeddings of each unique word as they appear in the corpus. A pooling operation is then used to distill a global word representation from all the unique words as a new word embedding. This process is made over the processing of the dataset and changes as the same word appears in other places.

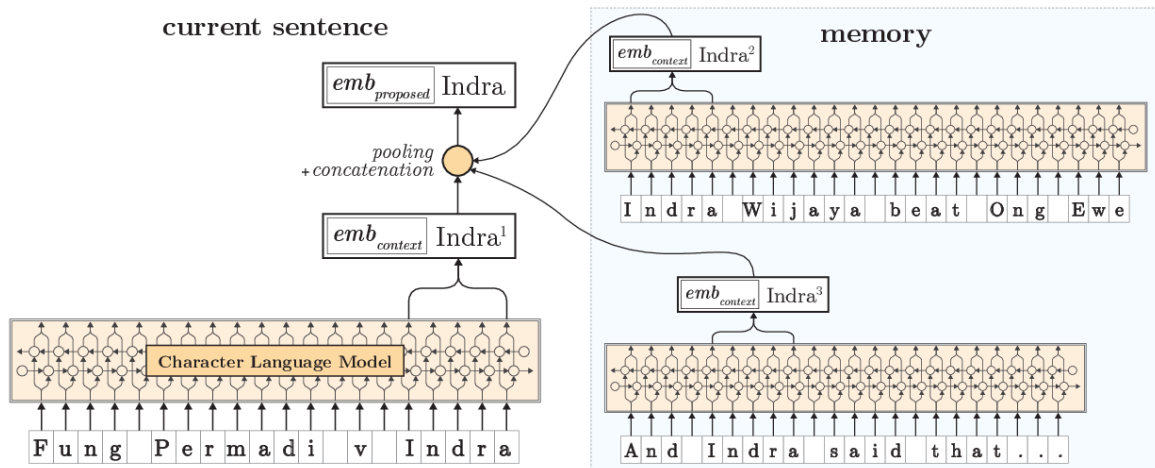


Figure 19: Example of the pooled Contextualized embedding generation [67].

With the new model, the authors achieved state-of-the-art scores for CONLL-03 NER in English and, surprisingly, in German. This model is particularly interesting for our case because it was tested for the task of **NER**.

2.5 SELECTED MODELS

In this Chapter, several models were contemplated making it clear which models should be used for our particular task. The models that are going to be compared are:

- **BERT**;
- DistilBERT;
- **XLM**;
- **XLM-R**;
- Pooled Contextualized Embeddings, also known as, Pooled Flair Embeddings.

All of this models, with the exception of the Pooled Contextualized Embeddings, are based on the principles of the Transformer model.

THE PROBLEM AND ITS CHALLENGES

During the Chapter 2 we have explored what is the current state of the art for the identified high level problem parts:

- Process the incoming [emails](#);
- Extract the relevant information from each [email](#);
- Create the quote document;
- Send the quote document.

In this Chapter 3, we will examine with more detail each sub problem and decide, where possible at this time, the best overall technological solution.

During this Chapter we will also present the system architecture within which it will be possible to observe where all parts are represented and how they fit together.

3.1 SMTP SERVER

A [Simple Mail Transfer Protocol \(SMTP\)](#) server will be needed to be used as per the illustration in Figure 5. The [SMTP](#) is a communication protocol, specifically for [email](#) transmission, being first defined as an internet standard in [RFC 821 \[4\]](#) and later expanded and clarified in [RFC 5321 \[35\]](#).

At this point, it was relevant to understand how was the current [email](#) system working in the company. It was gathered that the current address for quotations requests was an [email](#) alias created on the mail server.

Due to way the [SMTP](#) works and the current implementation of [email](#) services at the company it will not be possible to replace the current [SMTP](#) server with our implementation. In order to overcome this problem, several options were available, such as the creation of another [email](#) address and add it to the list of aliases of the current quotation [email](#) address, as per the [RFC 5321 \[35\]](#). Another option would be to setup a [SMTP](#) relay server, with authentication as to not be exploited by bad agents, where this server would then pipe the inbound [emails](#) to our service. One example of a software capable of this feature would be [Exim \[10\]](#).

Also per the illustration in Figure 5, a **SMTP** server is needed to send a response, in this case the quotation, back to the original sender, the customer. This can also be done in several ways, following the **RFC 821** [4] it is possible to use the **SMTP** server from our implementation, the current one from the company or even using a service for this kind of transactional email ¹.

Using an external service to send the response **email** has the benefit of decreasing the odds of the **email** being labelled as spam and protects sender reputation.

3.2 INFORMATION EXTRACTION

With the objective of extracting relevant information from the incoming **email** messages, the question that needs to be answer is what is the relevant information to extract from these **email** messages. The selected relevant information will then need to be translated into tags or labels to be used in the task of **NER**.

In order to train the models researched in Chapter 2 for the downstream task of **NER** the previously received quotation **email** messages will be needed. These **email** messages will also need to be tagged to create a dataset to train the models and evaluate them in order to choose, the best model for this specific problem.

Also in Chapter 2 the language models that presented the best results and behaviour for our purposes were:

- **Bidirectional Encoder Representations from Transformers (BERT)**;
- **DistilBERT**;
- **Cross-lingual Language Model (XLM)**;
- **XLM-R**;
- **Pooled Flair Embeddings**.

Each of the selected models may have several variants, for example, the **BERT** model has a cased and uncased version as well as a base and large model, so the relevant variants must also be included in the testing.

During an inquiry to the company with the intent of better understanding the problem, one of the conclusions was that one of the most relevant information to retrieve is what or which products are present in the **email** message. It is important to note that the products present in the **email** messages may or may not be available to buy from the company. This raises an important issue to address, how to search the current available products within the company.

¹ <https://postmarkapp.com/transactional-email>

3.3 QUOTE DOCUMENT

After finding and extracting the relevant information from one [email](#) message it is necessary to check if the product exists in the current products database.

When first approaching this issue, it might look simple, just use the company's database and perform a query to find the product in question. However it will not be possible to access the company's database, so the solution might be to create an ingress endpoint for the product information and store it on the service's own database.

The quote document will also need to be generated from the information found after the search. This document is currently generated by an employee on the company's [Enterprise resource planning \(ERP\)](#) in the form of a [PDF](#).

3.3.1 Database search

Having come to the conclusion that a database will be needed to store and search the products information, further research went into this topic.

The search resulted in two similar solutions, [Apache Solr](#) [28] and [Elasticsearch](#) [39]. Both solutions are based upon the free and open-source search engine software library [Apache Lucene](#) [20] and include a full-text search engine with an [HTTP](#) web interface and schema free [JavaScript Object Notation \(JSON\)](#) [58] documents.

Due to sharing the same underlying search engine, both [Apache Solr](#) and [Elasticsearch](#) have a similar high level way of working. They work by creating an index of documents, in this case, an index of products, where the schema of this documents would be the relevant parts used when searching for a product.

In the end, the choice between this two search engines came down to what was easier for the company to maintain. The company already uses [Amazon Web Services \(AWS\)](#) and for maintainability and billing reasons prefers, where possible, to use services provided by them. [AWS](#) provides a fully managed service for [Elasticsearch](#) ² and although [Elasticsearch](#) is currently changing it's license agreement ³, [AWS](#) still provides backing and support. The managed [Elasticsearch](#) service from [AWS](#) was, for these reasons, chosen as the search engine to use.

3.4 PROPOSED APPROACH - RISSA

Having a deeper understanding of all the requirements and a general approach on what technologies to use to achieve the desired functionalities, it is necessary to devise a system that encapsulates all of them.

² <https://aws.amazon.com/elasticsearch-service/>

³ <https://www.elastic.co/blog/licensing-change>

3.4.1 System architecture

As with most good projects, this system will also have a code name. The name chosen was Rissa, from the *Genus* of the kittiwake, a sea bird closely related with the seagull.

Following the previous Sections in this Chapter, it is now possible to illustrate the high level parts of the system in Figure 20, where the components between the dotted lines are the one's belonging to Rissa and the others, are the external components, either present on the company's premises or provided by [AWS](#).

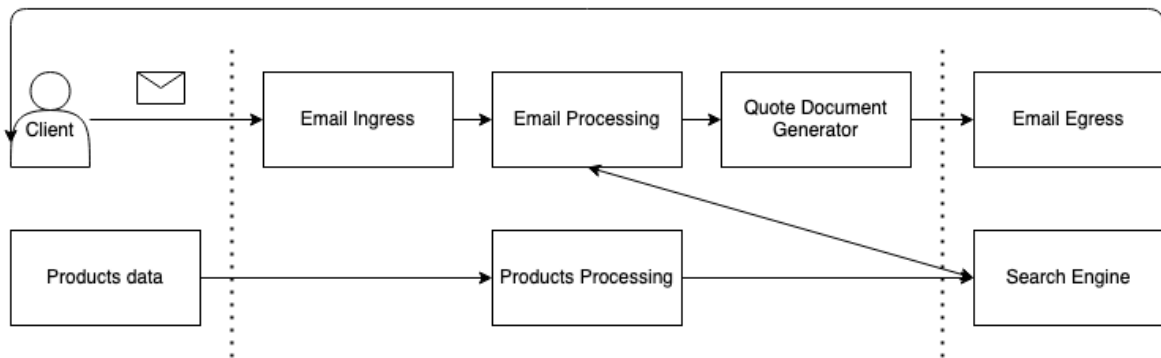


Figure 20: High level view of the system.

Starting from the services outside of Rissa, it will need a service program, represented by the *Products data* component, that exports the products information from the company's database server and uploads it directly to either the Elasticsearch service from [AWS](#), represented by the *Search Engine* component, or to Rissa. As to maintain a more unified process, it was decided that the upload would occur to Rissa' *Products Processing* component. This decision allows for the transparent change of the Elasticsearch service to another search service, thus removing the need to change the program more frequently. An important fact, since this will be deployed inside the company's network or even to the database server where maintenance costs are higher. This program will be ran within a predefined temporal routine as to not disrupt normal business hours functionality. This is not the ideal case as the products information will always be delayed in relation to the company's database server, a limitation imposed by not allowing outside access.

One other outside service will be the Elasticsearch service, the *Search Engine*, provided by [AWS](#). This service is fully managed and, as such, no deploy besides asking for an instance is necessary. The information to index from each product to the Elasticsearch will be the product's name, it's manufacturer code for search purposes and it's price for later use in the quote document generation.

The last outside service will be the outbound [SMTP](#) server, represented by the *Email Egress* component, as this was the best choice to ensure better [email](#) deliverability to the client. The

service chosen will be the [Amazon Simple Email Service \(SES\)](#) ⁴, also provided by [AWS](#) for the same reasons that led to the decision of the [Elasticsearch](#) managed service, in Section 3.3.1.

Getting to the Rissa application, and starting from the *Products Processing* component, it will be necessary to consume and process the uploaded data from the *Products data* component to then uploaded to the *Search Engine* component. For this part, a generic [HTTP Application Programming Interface \(API\)](#) will be created, simply as the ingress point delegating to another module the construction of the upload data to the *Search Engine* component, as per the [Elasticsearch](#) requirements.

Following the client's flow, the [email](#) message will first be received by the inbound [SMTP](#) server, represented by the *Email Ingress* component. This component is part of Rissa instead of being another external service, because it will allow for a finer control over the incoming [email](#).

After being received by the inbound [SMTP](#) server, the *Email Processing* component will extract all the necessary information. This component, later renamed as the information extraction module, will be composed by the [email](#) parser, responsible to extract and pre-process the [email](#)'s body content and the [NER](#) model that will tag the received text, in order to extract the relevant product information. The [NER](#) model will be inside it's own module as to be easier to change the model as the state of the art in the field of [NLP](#) advances. The information extract module will also be separated because, as was said earlier, although the focus will only be in [email](#) messages, it should be possible to grow Rissa to support more message services should the company desire.

The information extraction module will also be responsible for the search in the *Search Engine* component and, getting any tangible result, send the found products information to the next component. This behaviour makes this module the only other one to depend directly from the [Elasticsearch API](#), more concretely it's query system which making it the most vulnerable to outside changes.

Finally getting to the *Quote Document Generator* component, responsible to receive all the found products information and generating the quote document as a response to the client. This module depends on the [SES](#), and an abstraction should be made for easier service provider change.

Having described all the high level components of Figure 20, it's possible to observe Figure 21, a visual representation of the whole system's architecture.

⁴ <https://aws.amazon.com/ses/>

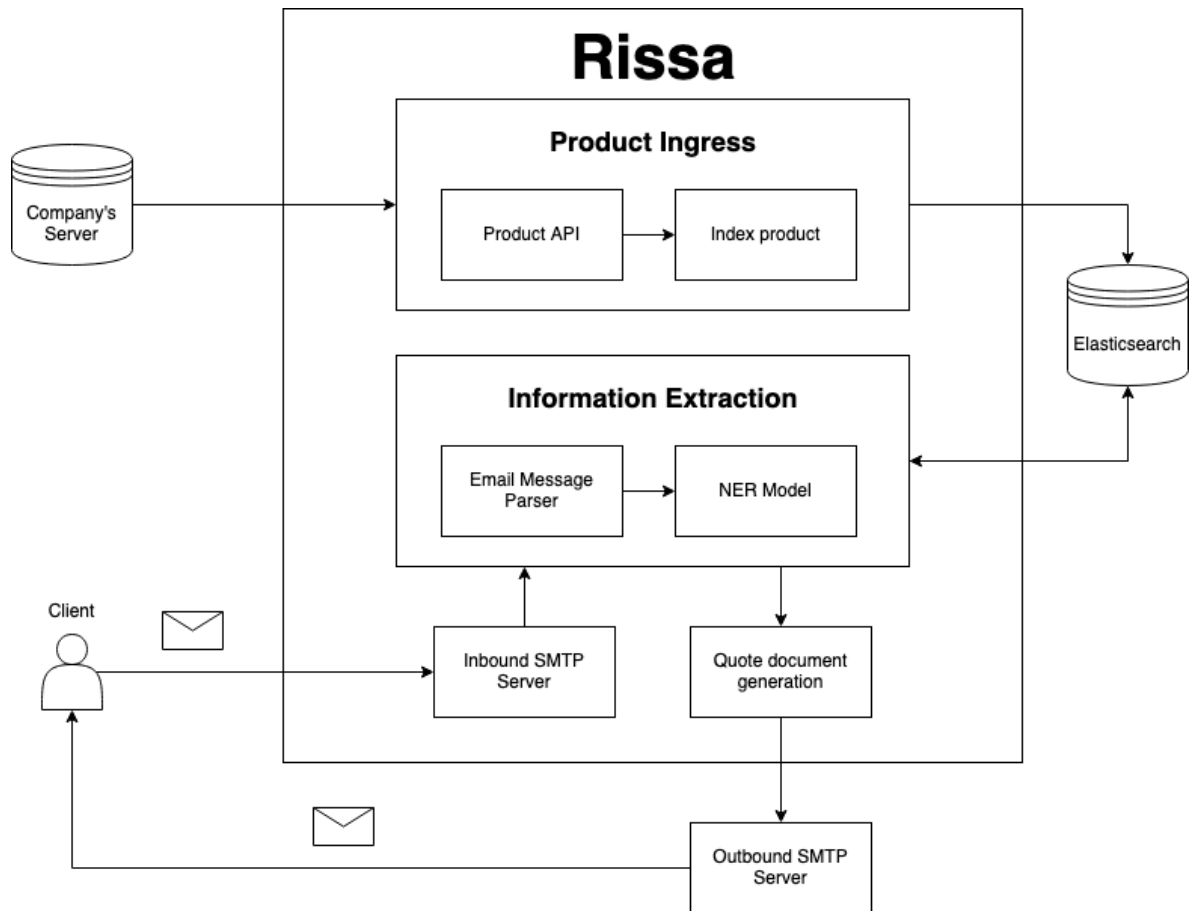


Figure 21: System Architecture.

3.4.2 Application technologies

Besides the functional requirements of Rissa, it should also be relatively easy to deploy, pointing in the direction of self contained compiled programming language. The gap, in the deployment, of interpreted or compiled programming languages has been decreasing due to the development of virtualisation and containerization technologies.

The programming language chosen for the development of Rissa was the Elixir [40] programming language due to it's functional, fault-tolerant and scalable capabilities. Elixir achieves fault-tolerance and scalability by leveraging the same runtime as the Erlang [5] programming language, the [Erlang Virtual Machine \(BEAM\)](#). By sharing the same runtime with Erlang, Elixir can use Erlang's code without any runtime cost, effectively taking advantage of the existing ecosystem.

For the development of Rissa, Elixir will be useful for the implementation of the inbound [SMTP](#) server, where it's shared nothing concurrent programming via message passing, also known as the Actor model [3, 24], will allow for scalability if necessary. This is particular

important because the handling of **email** is asynchronous and as such is more important to handle all the incoming messages than it is to have the lowest possible **email** response time.

Only one other component needs to be decided, the program that exports the products information from the company's database server and uploads it's information to Rissa. Although this component looks simple, it has some nuances that it will have to address. This program will be executed within a temporal interval, it's execution time should be reasonable as to not interfere with normal operation of the company's database. This leads to prefer languages that don't depend on a virtual machine during execution. Getting more specific on the company's database server, it was gathered that the environment is Windows Server [32] and the database is Microsoft **Structured Query Language (SQL)** Server [31]. With this information, the pool of available programming languages to choose from decreases, since it's preferable to have a programming languages that executes natively in the environment and has native drivers for the database.

The programming language chosen was the Go [37] programming language because it provides all the necessary requirements and is even possible to cross compile⁵ Go from other environments thus removing the need to have Windows Server present during development.

⁵ <https://github.com/golang/go/wiki/WindowsCrossCompiling>

DEVELOPMENT

In Chapter 3, we presented the system architecture and what technologies ... be used

This Chapter will explore what data is available, the important content of such data and the steps we will take to go from the raw data to a more usable format. Using this data we will find the best model for our task.

We will also develop the proposed solution, having, at the end, a system that tackles the proposed challenges.

4.1 DATASET

The dataset was made available, as several *eml* files. In this file format, each file contains one email in plain text in [MIME](#) format and all it's contents including attachments.

Each [email](#) present in the dataset was filtered beforehand by an employee, and was classified as a quotation request.

In order to process this files we use the Ruby [11] programming language with a mail library [33].

As the *eml* format is not well defined it is possible, in some cases, to have in one file the first email and then some or even all responses, where as in other cases the first email and the responses are in their individual files normally following the format `MSGID` for the first email and `MSGID-PART` for the subsequent, where the `MSGID` is an identifier for the email and `PART` is the number corresponding to next sequence.

After getting the general sense of the format of the data, and some of it's caveats, the next step was to start exploring concretely about it. The dataset is composed of 18650 email messages, of which 4190 are first contacts, that is, they are more likely are the initial quotation request, and as such will be where we will focus our attention. The time period of the dataset is of 324 days, taking place from 2019-11-06 to 2020-09-25.

In order to support the dissertation, it was also relevant to inspect how many quotation requests were resolved in the first reply. A request is regarded as resolved when no further exchange of emails occur in a conversation. To do this analysis we grouped the number of exchanged emails, between the employee and the client, by the number of times it occurred.

Figure 22 (note the logarithmic scale in the axis of ordinates) helps to visualize this information, and also shows that most of the interactions are resolved within the first reply. Getting

the concrete values observed in Table 3 shows that 70.29% of quotation requests were resolved in the first reply supporting the assumption that this first interaction has potential and benefit to be automated.

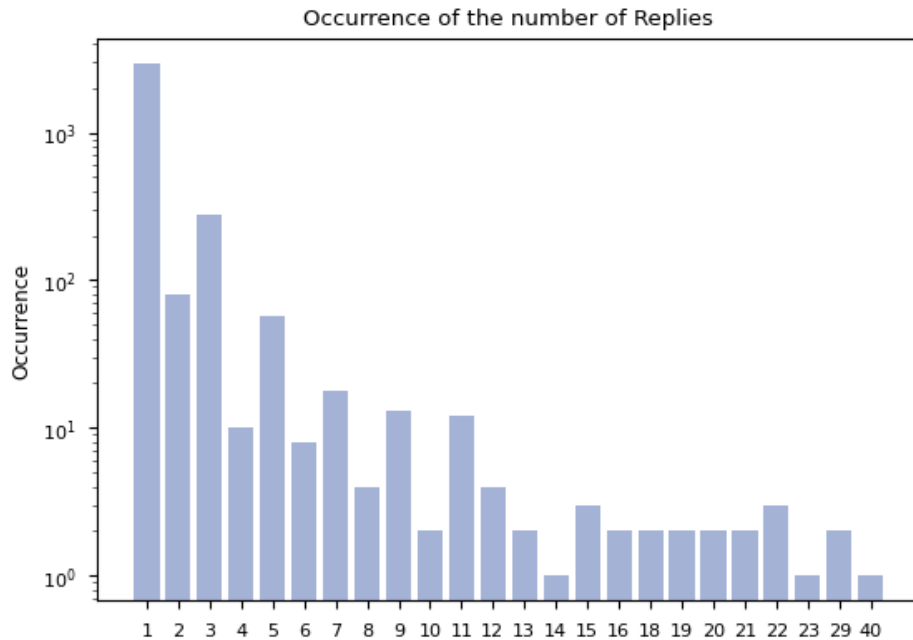


Figure 22: Exchanged [emails](#) in a conversation and it's occurrence.

The values in Table 3 do not add up to 100% because 734 emails do not have a reply. Upon further manual inspection and talks with the company, these emails were ignored because of their content. In their majority they were quotation requests for personal protective equipment due to the COVID-19 and SARS-CoV-2 pandemic.

4.1.1 Preprocessing

Having identified the [emails](#) that were most likely to have been resolved in the first reply, the next step was to separate the body content from the rest of the [email](#) fields. This preprocessing involved a two part process, each with it's sub steps.

Firstly the body content was extracted as raw text, effectively removing all the other [email](#) fields. In order to have more compatibility with future tools, all the text was converted from it's original encoding to [Unicode Transformation Format –8-bit \(UTF-8\)](#) [77]. With the help of another Ruby library, *Nokogiri* [34], [Hypertext Markup Language \(HTML\)](#) content present in the body was converted to text and replaced.

Secondly all whitespaces characters were addressed as some [emails](#) presented several *carriage return* characters which were removed. All *horizontal tabulations* were substituted for *spaces*. Ruby already has built-in regular expressions to handle this cases in the `Regexp` class, called *POSIX bracket expressions* that are similar to character classes but match any character

in the Unicode *Nd* category, for our case, [UTF-8](#). As such it is possible to apply them to our email body text, as seen in [Listing 1](#).

```
1 def squish(email_body)
2   email_body.gsub(/[[[:space:]]+/, " ").strip
3 end
```

Listing 1: Substitute all occurrences of one or more whitespaces for a common space.

To summarize, the [HTML](#) content present in the body was converted to text and the occurrences of one or more whitespaces were substituted by the most common space character in [UTF-8](#) (U+0020).

After this process all the resulting lines of text corresponding to each processed [email](#) were written to a file for the next steps. Having effectively created a new dataset from the original, the decision to use the simple format of a message per line, will prove itself useful as it facilitates the use of this new dataset in other tools.

4.1.2 *Manual entity tagging*

Having the new dataset, it is now necessary to annotate the emails in order to have a dataset for the models to train. In order to annotate the dataset, a key ingredient is missing, the entities that are important to identify. In other words, which is the relevant information present in the text that we want to extract.

To choose the most relevant information to identify and retrieve from the text, a deeper understanding of the dataset was needed. To gain more of this understanding, several [emails](#) were selected from the dataset and were read. This process also involved the company employees since, they have domain knowledge and helped validate or dismiss the assumptions made during the analysis. After some iterations it was settled on four entities:

- *SKU*: The internal or manufacturer code;
- *Product*: The name of the product;
- *Brand*: The brand name of the product;
- *Quantity*: The quantity of each product.

Having successfully identified the entities to use, the most manual labour inspection part follows, manually tagging each occurrence of each entity in each [email](#) text piece. The last part of the previous sentence, shows the iterative nature of the process, and hints at the possibility of increased automation.

Due to nature of this process a new tool had to be introduced. After some research, [doccano](#) [63], and open source text annotation tool, was chosen for several of it's features namely a

simple user interface, the possibility to interact with the system via an [API](#) that is [Representational State Transfer \(REST\)](#) [22] and easy to setup development environment.

Earlier it was hinted that the manual tagging process could use more automation. Instead of tagging all the messages upfront, it is possible to split them in several slices. Each slice, in our case contains *100 email* messages.

The tagging process is illustrated in [Figure 23](#), where for the first time, when a model is not available, the right side of the Figure is followed. In the right side the messages are uploaded to doccano and manually tagged being then exported to train the first model iteration. All the next iterations follow the left side, where firstly the messages are tagged with the current model and later corrected before being exported, just like in the right side, to train the next model iteration. The $model_{i-1}$ is compared with the $model_i$, where i is the current iteration, and the best performing model is kept. This process continues until no more slices are available.

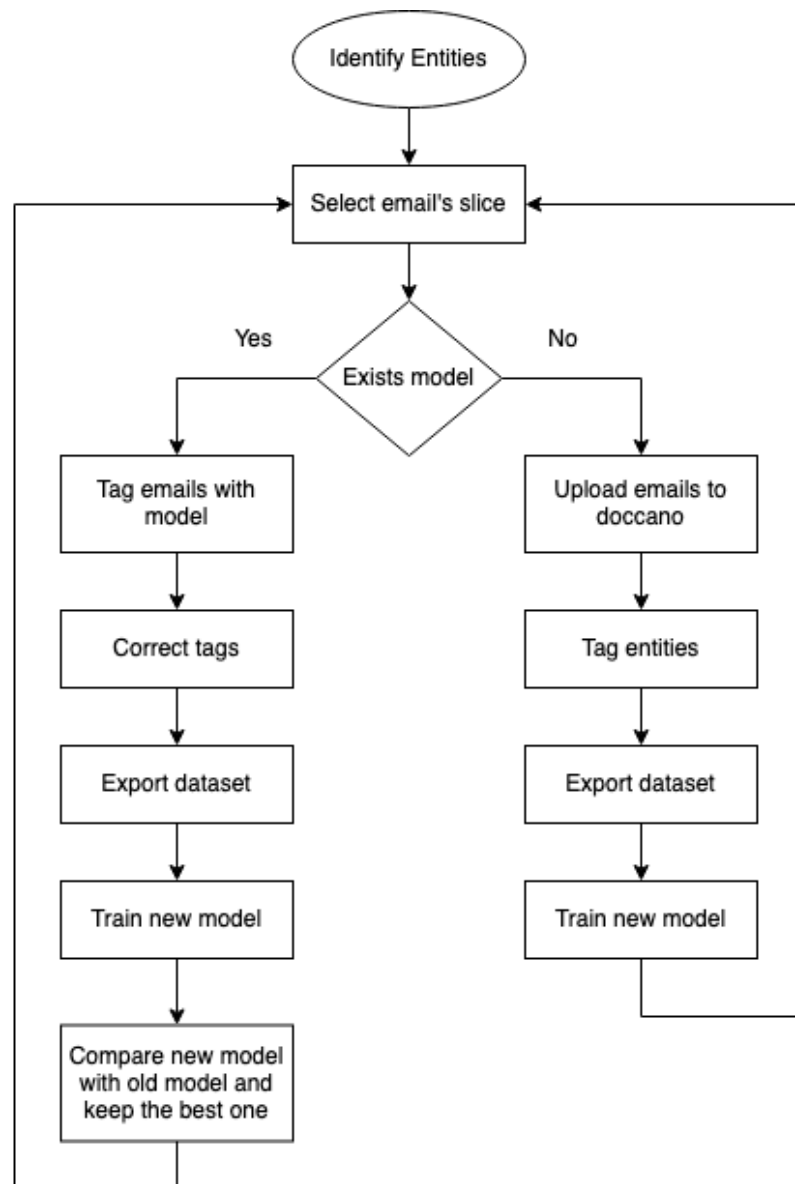


Figure 23: Illustration representing the tagging-training-correct flow.

The approach represented in Figure 23 can be faster or slower than the full manual approach (i.e. only using the right side of Figure 23), depending on the number of human resources manually tagging the messages and the available compute power to train the model.

The advantage of the presented approach is that after each slice it is possible to analyse in which cases the model is failing and change the tagging accordingly. For example, for this particular case, many costumers write the name of the product with the name of the brand. This makes it more difficult to identify whether to tag the with the label *Product* or the label *Brand*. In this case, when in doubt, it was chosen to favour the label *Product* has it was the one that showed more promising results.

As only some of the selected model presented in Chapter 2 could be used during this process, due to limitations of the computational resources, to use in the context of **Machine Learning (ML)**, only one model was used. The DistilBERT 2.4.5 model was chosen for its fast training time trait and for being based on BERT 2.4.5. In the future, and after the best base model was chosen, the DistilBERT model will be substituted by it in order to inspect it in between training slices and to improve it.

After some iterations of the process, more **emails** were discarded due to being in a language different than Portuguese as, for the time being, the only language to support is Portuguese. However this raised a concern about how to handle **emails** in other languages. Due to the experimental nature of the project and the vast majority of the **emails** being in Portuguese, this was pushed to a future feature.

In the end, 1008 **emails** messages were completely tagged with the result presented in Figure 24. This value is around a fourth of the initial value and took 11 process iterations to achieve.

In Figure 24 it is possible to visualize the impact of the earlier decision to give more prevalence to the *Product* tag instead of the *Brand* tag. This sample is also in line with the initial assumption that the most referred tags that refer to a product are the name (*Product* tag) and its synonymous or manufacturer code (*SKU* tag).

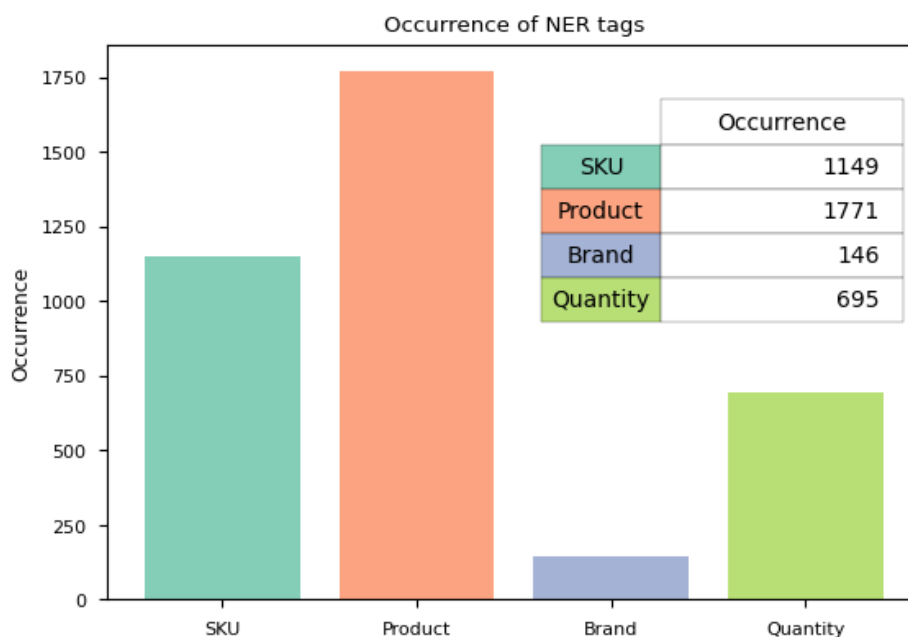


Figure 24: Occurrence of **NER** tags.

To conclude this Section, the manual tagging of text is still a time-consuming process, nevertheless it was possible to introduce a system to help speed up the process and at the same time inspect the performance between training slices.

4.2 TRAINING MODELS

During the tagging process and in order to use the DistilBERT model, it was needed to write the code that trained and applied the model to the next slice. For the sake of simplicity and code reuse, instead of implementing DistilBERT, the flair [62] library was chosen due to its simple API. In the case of Transformer models, flair uses HuggingFace's Transformers [78] which also provides an easy way to tap into a greater ecosystem of NLP libraries, such as TensorFlow [53] and PyTorch [55]. Both these libraries are implemented or provide bindings for Python [8] and as such the training part was written in Python.

Since many models use the CoNLL-2003 Shared Task [27] as a performance metric, the CoNLL-2003 format for the tagged text is a popular choice for the input. With this in mind, the dataset exported from doccano was converted to the CoNLL-2003 format.

As each chosen model is going to be tested with several options, several configuration files (Listing 2) were created, each configuration file representing a model and its options. The Python code abstracts each configuration file into an *Experiment dataclass* that represents it and an *ExperimentRunner* class that runs and saves the experiment for later analysis.

```

1 {
2     "description": "DistilBERT base cased with First Subword Pooling and
      ↪ Scalar Mix",
3     "embeddings": ["distilbert-base-multilingual-cased"],
4     "layers": [0,1,2,3,4,5,6],
5     "batch_size": 16,
6     "hidden_size": 256,
7     "max_epochs": 500,
8     "embeddings_storage_mode": "gpu",
9     "pooling_operation": "first",
10    "use_crf": true,
11    "use_scalar_mix": true,
12    "train_with_dev": false
13 }
```

Listing 2: Example of DistilBERT configuration file.

The models that are going to be trained and compared are:

- XLM model trained with [Masked Language Model \(MLM\)](#);
- [BERT Base Cased with First Subword Pooling and Scalar Mix](#);
- [BERT Large Cased with First Subword Pooling and Scalar Mix](#);

- Stacked Portuguese Word Embeddings with forward and backward Pooled Flair Embeddings;
- XLM-R Large with First and Last Subword Pooling and Scalar Mix;
- DistilBERT Base Cased with First Subword Pooling and Scalar Mix.

With the exception of the Pooled Flair Embeddings (Section 2.4.5), all other models are based on the principles of the Transformer model (Section 2.4.5) and also have the advantage of being multilingual capable.

4.2.1 Model results

During the training, the *Brand* label, due to its small number of occurrences, was not being learned satisfactorily and was removed for the final training.

In order to have a better reference, the models were executed three times. A new random seed was generated for each execution, but the same seed as maintained for all the models during training for that particular execution.

The results can be seen in Table 2 where the *Stacked Portuguese Word Embeddings with forward and backward Pooled Flair Embeddings* model has the advantage over the rest in the *Product* and *Quantity* labels and the *BERT Large Cased with First Subword Pooling and Scalar Mix* has advantage in the *SKU* label.

	Product	Quantity	SKU
XLM model trained with MLM (Masked Language Modeling)	61.08	30.55	68.85
BERT Base Cased with First Subword Pooling and Scalar Mix	37.44	31.64	65.34
BERT Large Cased with First Subword Pooling and Scalar Mix	56.11	43.7	72.99
Stacked Portuguese Word Embeddings with forward and backward Pooled Flair Embeddings	74.34	55.71	64.91
XLM-R Large with First and Last Subword Pooling and Scalar Mix	44.04	17.54	71.02
DistilBERT Base Cased with First Subword Pooling and Scalar Mix	42.84	28.31	64.3

Table 2: The mean F_1 score of every model for each chosen label.

Due to the prevalence of the *Product* label, as seen on Figure 24, it was chosen to use the *Stacked Portuguese Word Embeddings with forward and backward Pooled Flair Embeddings* model.

4.3 RISSA

Having found the most important part of the system, the model that is going to be used for the tasks of NER, the focus now switches over to the development of the system described in Section 3.4.1.

4.3.1 Product information ingress

Starting from the ingress of the product information, an [HTTP API](#) was created to handle the incoming collection. This single endpoint receives the entire products collection, because the total data transmitted is small, coming in under 5 [Megabyte \(MB\)](#). If the transmitted data were to be much larger, another protocol should be used instead of [HTTP](#). As the data is not big, using [HTTP](#) facilitates the development and future maintenance of the project.

The endpoint from Rissa is expecting a [HTTP POST](#) request with the body content as [JSON](#) [59] as exemplified in Listing 3.

```

1 {
2   "products": [
3     {
4       "internal_name": "Produto da MarcaA",
5       "sku": "MA-0001",
6       "price": 100
7     },
8     {
9       "internal_name": "Produto da MarcaB",
10      "sku": "MB-0001",
11      "price": 100
12    }
13  ]
14 }
```

Listing 3: Example of the body [HTTP](#) request to Rissa.

Since in [JSON](#) the only available datatypes for numbers are either an integer or a floating point and due to the way floating point precision works, as defined in [IEEE 754](#) [70], it was chosen to receive the price as an integer to avoid precision and rounding errors. This however, comes at the cost of not being able to represent more precision beyond cents or even more currencies besides the default chosen, the euro. To put things into perspective there are, as of [Unicode Common Locale Data Repository \(CLDR\)](#) version 38 [74], at least 59 currencies that don't use two decimal digits.

In order to protect Rissa from the rest of the internet an authentication system as implemented. This system involved the creation of an [API secret key](#). Rissa would then return to all requests made, to this endpoint, without the presence or a valid secret key the [HTTP error 401 - Unauthorized](#). As to not hardcode this secret key into the code, the introduction of a database besides [Elasticsearch](#) was needed. Since besides storing the secret keys, no other

requirement existed, for this reason, the database chosen was PostgreSQL [12] due to its familiarity and availability in AWS¹.

Having another database, specially one that features ACID transactions, the initial envisioned behaviour was altered to take advantage of this feature. When the request is made, all products are processed in a create or update fashion. As each product already features an unique internal code, this code is leveraged to create a new entry in the database if the code does not already exists, or to update an already existent entry with the incoming informations.

Both the secret API keys and the products entries are managed through Ecto [45], a library that provides data mapping from entries and/or queries to Elixir structs and also a Domain-specific Language (DSL) to write database queries.

```

1 defmodule Rissa.Shop.Product do
2   use Ecto.Schema
3
4   @primary_key {:id, :binary_id, autogenerate: true}
5   @foreign_key_type :binary_id
6   schema "products" do
7     field :internal_name, :string
8     field :sku, :string
9     field :price, :integer
10
11     field :inserted_at, :utc_datetime
12     field :updated_at, :utc_datetime
13   end
14 end

```

Listing 4: Representation of the *products* schema using Ecto.

After receiving the request, the products information is saved to PostgreSQL through the process described in Listing 4 and submitted to the Elasticsearch server through its API. As to not block the client making the request, after the products are saved to PostgreSQL, the server responds with HTTP code 201 - Created, terminating the interaction. Only after the request is completed, is the information submitted (Listing 5) to Elasticsearch, improving the overall reliability of the exportation process.

¹ <https://aws.amazon.com/rds/postgresql/>

```

1 {
2   "internal_name": "Produto da MarcaA",
3   "sku": "MA-0001"
4 }

```

Listing 5: Information indexed for each product in Elasticsearch.

In order to disrupt as little as possible the ability of Rissa to respond to **email** quote requests, a zero downtime index rebuilding process was implemented. To achieve this functionality, an alias index was created, with the name `rissa_products`. This index is an alias to the current index that follow the naming convention `rissa_products_timestamp`, where `timestamp` is the, local to the server Rissa is on, Unix Epoch time when the index started building. When a new index is finished building, the index alias changes from the current index to the newly created index and marks the old index for deletion. As such, in reality no index rebuilding occurs, only creation and deletion. This is also why no advantage is taken from the state provided by the `updated_at` field from each product entry in PostgreSQL.

Getting to the program that makes the requests and runs in the company's server, a database user and credentials were provided. A database view was also created, in order to minimize the tables that the user could access. To further ensure the security of the process, this user only has read permissions to the created view.

4.3.2 Inbound SMTP server

The implementation of the inbound **SMTP** server is a perfect example of the ability to leverage the Erlang ecosystem. The library that is going to be used is `gen_smtp`² which handles all the overhead of the protocol.

In Rissa it is only necessary to implement the feature behaviour `gen_smtp_server_session` of the library and pass the **email** data to the next module, as described in Listing 6.

```

1 defmodule Mail.SMTPServer do
2   @behaviour :gen_smtp_server_session
3   # ...
4   def handle_DATA(from, to, data, state) do
5     Mail.Receive.receive_message(from, to, data)
6     {:ok, UUID.uuid5(:dns, @domain_name, :default), state}
7   end
8   # ...
9 end

```

² https://github.com/gen-smtp/gen_smtp

Listing 6: Implementation of the `handle_DATA/4` callback from the behaviour `gen_smtp_server_session`.

Each request made to the inbound **SMTP** server is handled by a different **BEAM** process and is therefore isolated from and concurrent to the other processes.

4.3.3 Information extraction

When the information extraction module receives the **email**, it's preprocessing will be similar to preprocessing in Section 4.1 of the *eml* files. All whitespaces will be replaced with a single **UTF-8** space and the possible **HTML** content replace with it's plain text equivalent.

The preprocessed text is sent to the **NER** model for it to extract the relevant entities. The model that was chosen in Section 4.2.1 runs in a separate Python process and not on the **BEAM**. As such, a way to communicate from the **BEAM** to and from the Python process was needed. The **BEAM** provides several kinds of interoperability to achieve this functionality, being the main ones *Ports*, *C Nodes* or *Native Implemented Functions (NIFs)*. The safest interoperability is through *Ports* because *Ports* prevent the **BEAM** from crashing due to an error from the *Port*, in contrast to, for example, *NIFs*, where if the *NIF* code crashes, it also crashes the **BEAM**.

The Elixir module `Rissa.Ner` is responsible for spawning and managing the Python process. It does it with the help of `erlexec`³ which implements a manager of **Operating System (OS)** processes, an important aspect to prevent the occurrence of orphan processes.

Rissa communicates with the model process through **Standard Input (STDIN)** and **Standard Output (STDOUT)**. It sends to the **STDIN** of the model process a string containing the message and reads it's **STDOUT** in the form of **JSON** for easier processing. A minimal viable example is presented in Listing 7.

```

1 model = SequenceTagger.load(path)
2 model.eval()
3
4 def extract_entities(message):
5     sentence = Sentence(message)
6     model.predict(sentence)
7
8     entities = []
9     for entity in sentence.to_dict(tag_type='ner')['entities']:
10         labels = []
11         for label in entity['labels']:

```

³ <http://saleyn.github.io/erlexec/>

```

12         labels.append({
13             'label': label.value,
14             'score': round(label.score, 2)
15         })
16
17         entities.append({'text': entity['text'], 'labels': labels})
18
19     return entities
20
21
22 for line in fileinput.input():
23     print(json.dumps(extract_entities(line.rstrip())), flush=True)

```

Listing 7: Excerpt from the Python file that extract entities.

The module `Rissa.Ner` implements the *GenServer* behaviour (Listing 8) and is started in the beginning of the application. Ideally a `Rissa.Ner GenServer` would be started for each request, however only a single one is started, due to the high requirements of the [NER](#) model, both in term of [Random Access Memory \(RAM\)](#) usage and time to load.

```

1  defmodule Rissa.Ner
2    use GenServer
3    # ...
4    @impl true
5    def init(_state) do
6      state = :exec.run("python lib/ner/extract_ner.py", [:stdin, :stdout, :
7        ↪ monitor])
8      {:ok, state}
9    end
10   # ...
11 end

```

Listing 8: Function `init/1` for the `Rissa.Ner GenServer`.

For all the reasons stated above, the `Rissa.Ner` module will probably be the greatest bottleneck during execution.

After decoding the received [JSON](#), the matches are sorted accordingly to their score and extracted to be searched on the Elasticsearch server.

An example of the query used to search is presented in Listing 12. It works by querying the Elasticsearch index for either the product internal name or its internal code. As the search is simultaneous, in the case of the internal name, the query searches both for the exact name

match or tries to match other internal names, through fuzziness, with an error tolerance up until two characters. For the internal code, only exact matches are taken into consideration. The returned results are ordered by the score attributed by Elasticsearch.

Finally the collection of products resulted from the [email](#) message is passed to the quote document generation module where the quote document is created and sent to the client. This process is made by the `receive_message` function of the module `Mail.Receive` and is represented in Listing 9

```

1  defmodule Mail.Receive do
2    # ...
3    def receive_message(from, _to, data) do
4      Mail.Parsers.RFC2822.parse(data).body
5      |> String.replace(~r/[[:space:]]+/, " ")
6      |> Html.replace()
7      |> (&Rissa.Ner.extract_ner(Rissa.Ner, &1)).()
8      |> Jason.decode!()
9      |> Enum.map(&tuple_text_label/1)
10     |> Enum.map(&search_product/1)
11     |> Enum.reduce([], &reduce_products/2)
12     |> Email.products_quote(from)
13   end
14   # ...
15 end

```

Listing 9: [Email](#) message flow from after being received to before being sent.

4.3.4 Quote document generation

The quote document is sent as a [PDF](#) file, however due to the legal norms involved in creating and certifying the document, it would not be possible to implement the same functionality. Instead, Rissa will send a text [email](#) as a response, just to prove that the concept is feasible. In the future, it will only be needed to change the function that constructs the [email](#) to one that attaches the [PDF](#) document, as seen in Listing 10.

```

1 defmodule Rissa.Email do
2   # ...
3   def products_quote(products, to) do
4     base_email()
5     |> to(to)
6     |> subject(opts().subject)
7     |> put_header("Reply-To", opts().reply_to)
8     # |> put_attachment(create_quote_pdf(products))
9     |> render(:products_quote, products: products)
10  end
11  # ...
12 end

```

Listing 10: Quote `email` response construction function with the option to also add a `PDF` file.

In Section 4.3.1 it was referred that the price of each product was being stored as an integer. In order to represent the price to the client, some formatting needed to be done. Firstly we used a library ⁴ that provided arbitrary precision decimal arithmetic to add back the precision by dividing by 100. Secondly and following the `CLDR` format, a library ⁵ that implemented it was used as to output a usable string, as can be observed in Listing 11.

```

1 defmodule RissaWeb.EmailView do
2   use RissaWeb, :view
3
4   def humanize_money(product) do
5     product.price
6     |> Decimal.div(100)
7     |> Rissa.Cldr.Number.to_string(locale: "pt-pt", currency: "EUR")
8   end
9 end

```

Listing 11: Convert the integer product price to a human friendly string.

4.3.5 Benchmarks

Having developed all the components of `Rissa`, it is now possible to evaluate the hypothesis raised in Section 4.3.3 and identify the biggest run time bottleneck.

⁴ <https://github.com/ericmj/decimal>

⁵ <https://github.com/elixir-cldr/cldr>

The function present in Listing 9 was decomposed in three subparts, the pre-process, the `extract_ner` and the search part. All these subparts were run with the same initial inputs and the same warm up and execution time parameters. In order to remove network latencies from the search part, this benchmark was run with the PostgreSQL and Elasticsearch services in the same machine.

A graphical representation, presented in Figure 25, of the benchmark confirms the hypothesis. The benchmark also reveals that the only non concurrent part, the `extract_ner` of the system take on average 0.88 seconds to complete. This information is relevant, as it allows for an estimate of the throughput of the system, i.e. the throughput of handle `email` messages per time period.

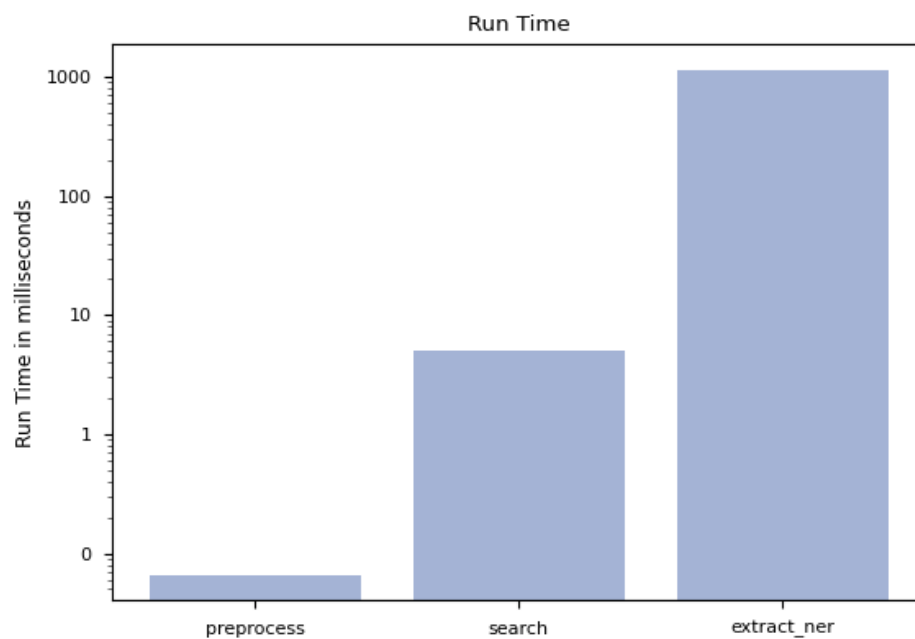


Figure 25: Run time of the three main parts of the function present in listing 9.

4.4 SUMMARY

During this Chapter the dataset provided was explored and processed until the information was fit for the model training. A text tagging system was also created, in order to help the process of labelling the `email` data. After the labelling of the data, several models with different options were trained and as a culmination, the best model, for our case, was found.

All the different parts of the proposed solution were addressed and developed and, in the end, the system developed closely resembles the proposed system architecture.

CASE STUDY

5.1 EXPERIMENTAL SETUP

Following the same trend as the other services, Rissa was deployed in an Amazon EC2 instance. In order to choose the most appropriate instance for the work load, measurements had to be taken.

The adjacent `NER` model of the `Rissa.Ner` module was profiled in terms of its memory usage and the results can be seen in Figure 26.

The model, takes on average less than 20 seconds to load and requires 10 `GibiByte (GiB)` of memory at peak time before stabilising at 4.68 `GiB` after the `Garbage Collection (GC)` process.

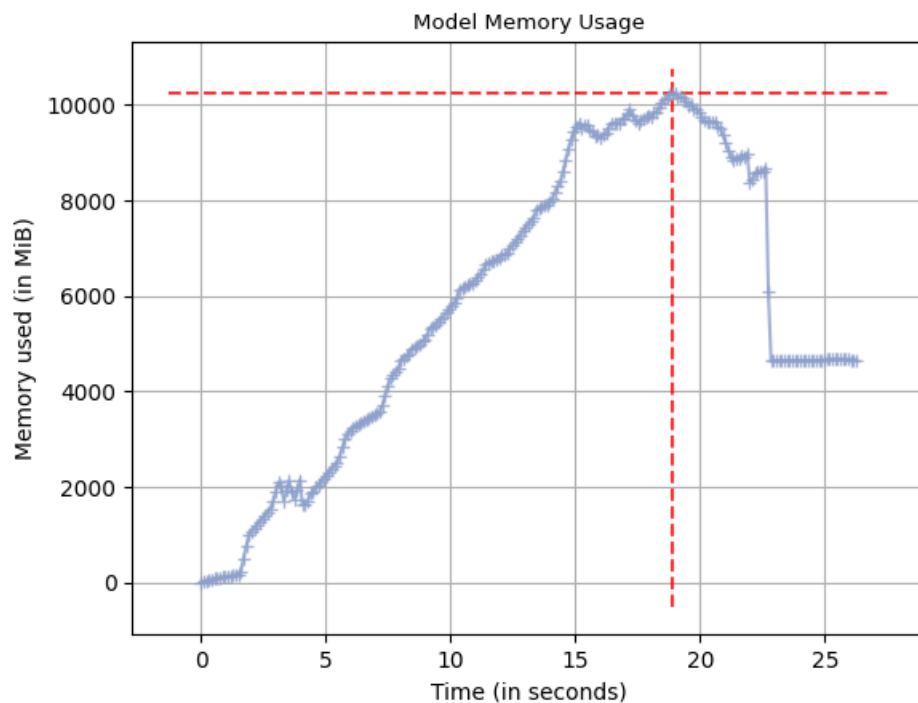


Figure 26: Model memory usage per second, during initialization time.

After both these results and the results presented in Section 4.3.5, it was confirmed that the module `Rissa.Ner` is the greatest bottleneck both during execution and load time.

Giving this minimum requirements, the instance chosen was the *t3.xlarge* with 4 **Virtual Central Processing Unit (vCPU)** and 16 **GiB of RAM**. This instance was chosen instead of the smaller and cheaper *t3.large* with 2 **vCPU** and 8 **GiB of RAM**, because it presents a better price/performance relation ¹.

Having decided on the instance it is now possible to stress test the system in order to establish it's capacity. To help with this task, the tool *Postal* [30] was used. This test was executed during 60 minutes and it's results are presented in Figure 27. The system averages 108 processed **emails** per minute, confirming the earlier results presented in Figure 25.

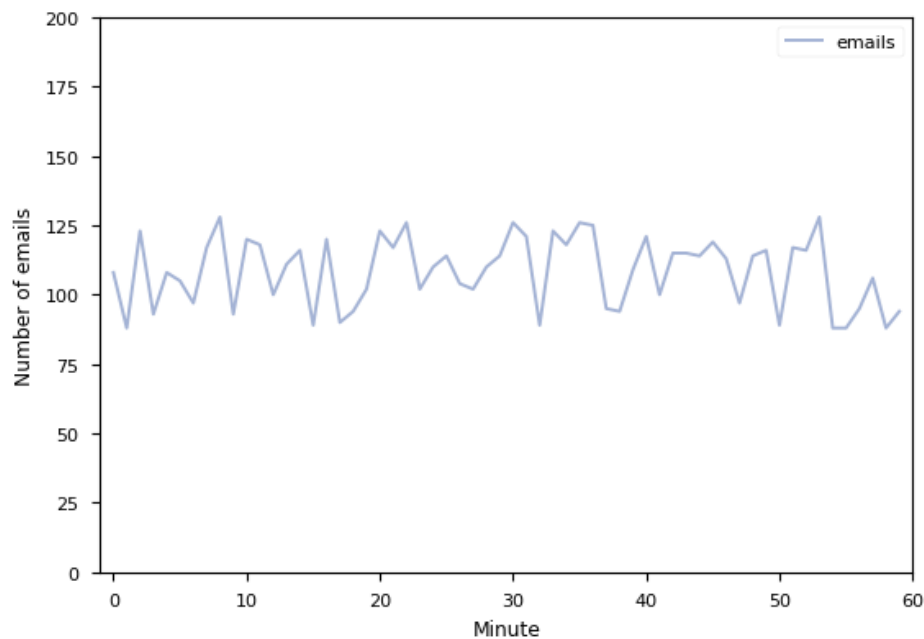


Figure 27: Processed **email** messages per minute.

5.2 RESULTS

Having the system deployed, an alias **email** address was given to Rissa in order for it to receive the same **email** messages as the quotes **email** address. This setup ran during 30 days, as to have a real insight into the operation and results of the proposed solution.

During the time that Rissa ran, it received 548 **email** messages and all the **email** responses were logged to be analysed. The employees **email** responses were also obtained, in order to compare then to Rissa's results.

Following the same analysis made in Section 4.1, only the **email** messages that were resolved in the first reply were taken into consideration, leaving 372 **email** messages to analyse.

¹ <https://calculator.aws>

Instead of labelling the remaining [email](#) messages, only the response was taken into account in order to collect what products were present in the quote document sent as a response.

After associating the products with each corresponding [email](#) message, the employees' responses were compared with those of Rissa. Rissa achieved 8.6% of full accuracy, meaning that 32 responses were the same as those sent by the employees, and 20.97% of partial accuracy, meaning that 78 responses had one or more correct products present but not all.

Upon recreating the sending of the [email](#) messages and inspecting both the results from the [NER](#) model and the subsequent search, the reasons for the above results became clear. Some products references are not being labelled by the [NER](#) model, as expected by its F_1 score, but the search is where the main problem is occurring. The product chosen by Elasticsearch most of the time is not the correct one because many of the labelled text that is being sent to search, doesn't provide enough context.

Although the search didn't provide good results, it made a great job of eliminating the false positives that it received from the [NER](#) model.

5.3 DISCUSSION

A more detailed analysis of the answers given by Rissa in relation to those given by the employees revealed a discrepancy in the answers for the same type of product, i.e. Rissa gives a similar answer for a product to that given by the employees. To better understand this difference in answers, an interview was conducted in the company where the difference in criteria in the decision to choose a product was perceived. In addition to the information made available to Rissa, employees have access to internal information about repeated customers, such as purchase history and also more volatile information such as internal stock availability and availability of stock from the manufacturer.

From this knowledge it was possible to arrive at a hypothesis of why the discrepancy in responses between Rissa and the employees occurred.

Having the opportunity to test the system in the real world and with real information was immensely helpful as it allowed to show what parts of the system were up to the task and what parts need improvement.

The developed and deployed system works without disturbances or visible impact to both the internal and external company's [Information Technology \(IT\)](#) infrastructures.

CONCLUSION

6.1 CONCLUSIONS

The work described in this dissertation took place in Uilmédica, a medical and hospital products business company. This dissertation, attempts and almost completely succeeds in solving a real world problem of the company. The problem presented by the company was if it was possible to automatically answer to quotation [email](#) requests.

This dissertation started by searching for market tools that could solve the problem and didn't found such tool. Having not identified a viable market solution, we start by identifying the main parts of the problem and proceed to research what is the current state of the art for each relevant one.

After getting a better understanding of what is currently possible to achieve in each determined field, the work proceeds to a deeper analyses of the problem and it's challenges. It is during this phase that most decisions regarding the overall design of the system start to occur culminating in the presentation of the system's architecture and the technologies that used to develop it.

Having the system developed, it was deployed during a period of time to better understand it's behaviour in real situations. During this period, several benchmarks were also performed and the system's limits tested.

The extraction of useful data from unstructured text is still a difficult task to do but the implementation of Rissa shows that it is possible to implement a system that uses state of the art [NLP](#) research work in a real world situation.

In this sense, it is possible to conclude that the work developed and presented through this dissertation is applicable in the context of this company, in such a way that it can improve the efficiency of the [email](#) quotation response process.

6.2 FUTURE WORK

During the research and the development of the system, several possible improvements emerged.

Although the main language presented in the messages was Portuguese, a minority of the messages were in other languages. The model that was chosen was the only model that wasn't multilingual. To be able to extract **NER** from other languages either the model is changed to one that offers multilingual support, or it is necessary to detect first the language and then use a different model for each language.

Since the greatest bottleneck in the capacity to process messages in Rissa is the model, when necessary, more model processes could be executed, at the expense of more memory used, to further increase Rissa's overall throughput

The focus of this work was on handling **email**, but Rissa is sufficiently modular to be easier used with other messaging services.

By changing Rissa's **NER** model to another model that has been trained on another dataset, it is possible to use the rest of the system, in other companies from other areas, to do similar work.

BIBLIOGRAPHY

- [1] Wilson L. Taylor. ‘ “Cloze Procedure” : A New Tool for Measuring Readability’. In: *Journalism Quarterly* 30.4 (1953), pp. 415–433. DOI: [10 . 1177 / 107769905303000401](https://doi.org/10.1177/107769905303000401). eprint: <https://doi.org/10.1177/107769905303000401>. URL: <https://doi.org/10.1177/107769905303000401>.
- [2] Zellig S. Harris. ‘Distributional Structure’. In: *WORD* 10.2-3 (1954), pp. 146–162. DOI: [10.1080/00437956.1954.11659520](https://doi.org/10.1080/00437956.1954.11659520). URL: <https://doi.org/10.1080/00437956.1954.11659520>.
- [3] Carl Hewitt, Peter Bishop and Richard Steiger. ‘A Universal Modular ACTOR Formalism for Artificial Intelligence’. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. IJCAI’73*. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.
- [4] Jonathan B. Postel. *SIMPLE MAIL TRANSFER PROTOCOL*. Aug. 1982. URL: <https://tools.ietf.org/html/rfc821> (visited on 7th Jan. 2020).
- [5] Joe Armstrong. *Erlang Programming Language*. 1986. URL: <https://www.erlang.org/> (visited on 20th Jan. 2021).
- [6] Warren S. McCulloch and Walter Pitts. ‘A Logical Calculus of the Ideas Immanent in Nervous Activity’. In: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, pp. 15–27. ISBN: 0262010976.
- [7] Jeffrey L. Elman. ‘Finding structure in time’. In: *Cognitive Science* 14.2 (1990), pp. 179–211. ISSN: 0364-0213. DOI: [https://doi.org/10.1016/0364-0213\(90\)90002-E](https://doi.org/10.1016/0364-0213(90)90002-E). URL: <http://www.sciencedirect.com/science/article/pii/036402139090002E>.
- [8] Guido van Rossum. *Python Programming Language*. 1992. URL: <https://www.python.org/> (visited on 10th Dec. 2020).
- [9] Corinna Cortes and Vladimir Vapnik. ‘Support-Vector Networks’. In: *Mach. Learn.* 20.3 (Sept. 1995), pp. 273–297. ISSN: 0885-6125. DOI: [10.1023/A:1022627411411](https://doi.org/10.1023/A:1022627411411). URL: <https://doi.org/10.1023/A:1022627411411>.
- [10] Philip Hazel. *Exim Internet Mailer*. 1995. URL: <https://www.exim.org/> (visited on 20th Jan. 2021).
- [11] Yukihiro Matsumoto. *Ruby Programming Language*. 1995. URL: <https://www.ruby-lang.org/> (visited on 10th Dec. 2020).
- [12] PostgreSQL Global Development Group. *PostgreSQL: The World’s Most Advanced Open Source Relational Database*. July 1996. URL: <https://www.postgresql.org/> (visited on 8th Dec. 2020).

- [13] K. Moore. *Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text*. Nov. 1996. URL: <https://tools.ietf.org/html/rfc2047> (visited on 7th Jan. 2020).
- [14] J. Postel N. Freed J. Klensin. *Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*. Nov. 1996. URL: <https://tools.ietf.org/html/rfc2048> (visited on 7th Jan. 2020).
- [15] N. Borenstein N. Freed. *Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples*. Nov. 1996. URL: <https://tools.ietf.org/html/rfc2049> (visited on 7th Jan. 2020).
- [16] N. Borenstein N. Freed. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. Nov. 1996. URL: <https://tools.ietf.org/html/rfc2045> (visited on 7th Jan. 2020).
- [17] N. Borenstein N. Freed. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. Nov. 1996. URL: <https://tools.ietf.org/html/rfc2046> (visited on 7th Jan. 2020).
- [18] Sepp Hochreiter and Jürgen Schmidhuber. ‘Long Short-Term Memory’. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. eprint: <https://doi.org/10.1162/neco.1997.9.8.1735>. URL: <https://doi.org/10.1162/neco.1997.9.8.1735>.
- [19] F. Cummins. ‘Learning to forget: continual prediction with LSTM’. English. In: *IET Conference Proceedings* (Jan. 1999), 850–855(5). URL: https://digital-library.theiet.org/content/conferences/10.1049/cp_19991218.
- [20] Apache Software Foundation Doug Cutting. *Apache Lucene*. 1999. URL: <https://lucene.apache.org/> (visited on 20th Jan. 2021).
- [21] Microsoft Corporation. *C Sharp (C#)*. 2000. URL: <https://docs.microsoft.com/dotnet/csharp/> (visited on 20th Jan. 2021).
- [22] Roy Thomas Fielding. ‘Architectural Styles and the Design of Network-based Software Architectures’. Doctoral dissertation. University of California, Irvine, 2000, pp. 76–106. URL: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [23] J.S. Cramer. ‘The Origins of Logistic Regression’. In: *Tinbergen Institute, Tinbergen Institute Discussion Papers* (Jan. 2002). DOI: 10.2139/ssrn.360300.
- [24] Joe Armstrong. ‘Making reliable distributed systems in the presence of software errors’. Doctoral dissertation. KTH Royal Institute of Technology, Sweden, 2003. URL: https://erlang.org/download/armstrong_thesis_2003.pdf.
- [25] Yoshua Bengio et al. ‘A Neural Probabilistic Language Model’. In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

- [26] Xavier Carreras, Lluís Màrquez and Lluís Padró. ‘A Simple Named Entity Extractor Using AdaBoost’. In: *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003 - Volume 4*. CONLL ’03. Edmonton, Canada: Association for Computational Linguistics, 2003, pp. 152–155. DOI: [10.3115/1119176.1119197](https://doi.org/10.3115/1119176.1119197). URL: <https://doi.org/10.3115/1119176.1119197>.
- [27] Erik F. Tjong Kim Sang and Fien De Meulder. ‘Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition’. In: *CoRR* cs.CL/0306050 (2003). URL: <http://arxiv.org/abs/cs/0306050>.
- [28] Apache Software Foundation Yonik Seeley. *Apache Solr*. 2004. URL: <https://lucene.apache.org/solr/> (visited on 20th Jan. 2021).
- [29] Cristian Buciluundefined, Rich Caruana and Alexandru Niculescu-Mizil. ‘Model Compression’. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’06. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 535–541. ISBN: 1595933395. DOI: [10.1145/1150402.1150464](https://doi.org/10.1145/1150402.1150464). URL: <https://doi.org/10.1145/1150402.1150464>.
- [30] Russell Coker. *Postal*. 2006. URL: <https://doc.coker.com.au/projects/postal/>.
- [31] Microsoft Corporation. *Microsoft SQL Server*. 2007. URL: <https://www.microsoft.com/sql-server/> (visited on 20th Jan. 2021).
- [32] Microsoft Corporation. *Windows Server*. 2007. URL: <https://www.microsoft.com/windows-server> (visited on 20th Jan. 2021).
- [33] Mikel Lindsaar. *A Really Ruby Mail Library*. 2008. URL: <https://github.com/mikel/mail> (visited on 10th Dec. 2020).
- [34] Sparkle Motion. *Nokogiri (鋸) is an HTML, XML, SAX, and Reader parser*. 2008. URL: <https://github.com/sparklemotion/nokogiri> (visited on 10th Dec. 2020).
- [35] J. Klensin Network Working Group. *Simple Mail Transfer Protocol*. Oct. 2008. URL: <https://tools.ietf.org/html/rfc5321> (visited on 7th Jan. 2020).
- [36] Ed. P. Resnick. *Internet Message Format*. Oct. 2008. URL: <https://tools.ietf.org/html/rfc5322> (visited on 7th Jan. 2020).
- [37] Ken Thompson Robert Griesemer Rob Pike. *The Go Programming Language*. 2009. URL: <https://golang.org/> (visited on 20th Jan. 2021).
- [38] Tomas Mikolov et al. ‘Recurrent neural network based language model’. In: vol. 2. Jan. 2010, pp. 1045–1048.
- [39] Elastic NV Shay Banon. *Elasticsearch*. Feb. 2010. URL: <https://www.elastic.co/elasticsearch/> (visited on 20th Jan. 2021).
- [40] José Valim. *The Elixir programming language*. 2011. URL: <https://elixir-lang.org/> (visited on 20th Jan. 2021).

- [41] György Móra and Veronika Vincze. ‘Joint Part-of-Speech Tagging and Named Entity Recognition Using Factor Graphs’. In: *Text, Speech and Dialogue*. Ed. by Petr Sojka et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 232–239. ISBN: 978-3-642-32790-2.
- [42] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. arXiv: [1310.4546](https://arxiv.org/abs/1310.4546) [cs.CL].
- [43] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: [1301.3781](https://arxiv.org/abs/1301.3781) [cs.CL].
- [44] Ilya Sutskever. ‘Training Recurrent Neural Networks’. PhD thesis. CAN, 2013. ISBN: 9780499220660.
- [45] José Valim. *Ecto: A toolkit for data mapping and language integrated query*. July 2013. URL: <https://hexdocs.pm/ecto/Ecto.html> (visited on 8th Dec. 2020).
- [46] Cicero Dos Santos and Bianca Zadrozny. ‘Learning Character-level Representations for Part-of-Speech Tagging’. In: vol. 5. July 2014.
- [47] Jeffrey Pennington, Richard Socher and Christopher D. Manning. *GloVe: Global Vectors for Word Representation*. Aug. 2014. URL: <https://nlp.stanford.edu/projects/glove/> (visited on 13th Jan. 2020).
- [48] Jeffrey Pennington, Richard Socher and Christopher D. Manning. ‘GloVe: Global Vectors for Word Representation’. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: <http://www.aclweb.org/anthology/D14-1162>.
- [49] Klaus Greff et al. ‘LSTM: A Search Space Odyssey’. In: *CoRR abs/1503.04069* (2015). arXiv: [1503.04069](https://arxiv.org/abs/1503.04069). URL: <http://arxiv.org/abs/1503.04069>.
- [50] Geoffrey Hinton, Oriol Vinyals and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: [1503.02531](https://arxiv.org/abs/1503.02531) [stat.ML].
- [51] Yoon Kim et al. *Character-Aware Neural Language Models*. 2015. arXiv: [1508.06615](https://arxiv.org/abs/1508.06615) [cs.CL].
- [52] Cicero Nogueira dos Santos and Victor Guimarães. *Boosting Named Entity Recognition with Neural Character Embeddings*. 2015. arXiv: [1505.05008](https://arxiv.org/abs/1505.05008) [cs.CL].
- [53] Google Brain Team. *TensorFlow*. 2015. URL: <https://www.tensorflow.org/> (visited on 10th Dec. 2020).
- [54] Dzmitry Bahdanau, Kyunghyun Cho and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: [1409.0473](https://arxiv.org/abs/1409.0473) [cs.CL].
- [55] Facebook Artificial Intelligence Research Lab. *PyTorch*. 2016. URL: <https://pytorch.org/> (visited on 10th Dec. 2020).
- [56] Rico Sennrich, Barry Haddow and Alexandra Birch. *Neural Machine Translation of Rare Words with Subword Units*. 2016. arXiv: [1508.07909](https://arxiv.org/abs/1508.07909) [cs.CL].
- [57] Vered Shwartz. *Representing Words*. Jan. 2016. URL: <http://veredshwartz.blogspot.com/2016/01/representing-words.html> (visited on 13th Jan. 2020).

- [58] Ecma International. *ECMA-404, The JSON data interchange syntax, 2nd edition*. Dec. 2017. URL: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.
- [59] Ed. T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Dec. 2017. URL: <https://tools.ietf.org/html/rfc8259> (visited on 7th Jan. 2020).
- [60] Ashish Vaswani et al. ‘Attention Is All You Need’. In: *CoRR abs/1706.03762* (2017). arXiv: [1706.03762](https://arxiv.org/abs/1706.03762). URL: <http://arxiv.org/abs/1706.03762>.
- [61] Tom Young et al. ‘Recent Trends in Deep Learning Based Natural Language Processing’. In: *CoRR abs/1708.02709* (2017). arXiv: [1708.02709](https://arxiv.org/abs/1708.02709). URL: <http://arxiv.org/abs/1708.02709>.
- [62] Alan Akbik, Duncan Blythe and Roland Vollgraf. ‘Contextual String Embeddings for Sequence Labeling’. In: *COLING 2018, 27th International Conference on Computational Linguistics*. 2018, pp. 1638–1649.
- [63] Hiroki Nakayama et al. *doccano: Text Annotation Tool for Human*. Software available from <https://github.com/doccano/doccano>. 2018. URL: <https://github.com/doccano/doccano>.
- [64] Matthew E. Peters et al. ‘Deep contextualized word representations’. In: *CoRR abs/1802.05365* (2018). arXiv: [1802.05365](https://arxiv.org/abs/1802.05365). URL: <http://arxiv.org/abs/1802.05365>.
- [65] A. Radford. ‘Improving Language Understanding by Generative Pre-Training’. In: 2018.
- [66] Elvis Saravia. *Modern Deep Learning Techniques Applied to Natural Language Processing*. 2018. URL: https://github.com/omarsar/nlp_overview (visited on 22nd Feb. 2021).
- [67] Alan Akbik, Tanja Bergmann and Roland Vollgraf. ‘Pooled Contextualized Embeddings for Named Entity Recognition’. 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)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 724–728. DOI: [10.18653/v1/N19-1078](https://doi.org/10.18653/v1/N19-1078). URL: <https://www.aclweb.org/anthology/N19-1078>.
- [68] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: [1810.04805](https://arxiv.org/abs/1810.04805) [cs.CL].
- [69] Gurbinder Gill et al. *Distributed Word2Vec using Graph Analytics Frameworks*. 2019. arXiv: [1909.03359](https://arxiv.org/abs/1909.03359) [cs.LG].
- [70] IEEE. ‘IEEE-754, Standard for Floating-Point Arithmetic’. In: *IEEE Std 754-2019* (June 2019).
- [71] Guillaume Lample and Alexis Conneau. *Cross-lingual Language Model Pretraining*. 2019. arXiv: [1901.07291](https://arxiv.org/abs/1901.07291) [cs.CL].
- [72] Alex Wang et al. ‘GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding’. In: *In the Proceedings of ICLR*. 2019.

- [73] Alexis Conneau et al. *Unsupervised Cross-lingual Representation Learning at Scale*. 2020. arXiv: [1911.02116](https://arxiv.org/abs/1911.02116) [cs.CL].
- [74] Unicode Consortium. *CLDR 38 Release Note - CLDR - Unicode Common Locale Data Repository*. Oct. 2020. URL: <http://cldr.unicode.org/index/downloads/cldr-38> (visited on 7th Dec. 2020).
- [75] Matthew Honnibal et al. *spaCy: Industrial-strength Natural Language Processing in Python*. 2020. DOI: [10.5281/zenodo.1212303](https://doi.org/10.5281/zenodo.1212303). URL: <https://doi.org/10.5281/zenodo.1212303>.
- [76] Victor Sanh et al. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2020. arXiv: [1910.01108](https://arxiv.org/abs/1910.01108) [cs.CL].
- [77] The Unicode Consortium. *The Unicode Standard*. Tech. rep. Version 13.0.0. Unicode Consortium, 2020. URL: <https://www.unicode.org/versions/Unicode13.0.0/>.
- [78] Thomas Wolf et al. ‘Transformers: State-of-the-Art Natural Language Processing’. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. URL: <https://www.aclweb.org/anthology/2020.emnlp-demos.6>.
- [79] Parseur B.V. *Powerful email parser software | Parseur*. URL: <https://parseur.com/> (visited on 30th Jan. 2020).
- [80] Liddy E.D. 2001. ‘Natural Language Processing’. In: *Encyclopedia of Library and Information Science*, 2nd Ed. NY. Marcel Decker, Inc.
- [81] Matthew Guay. *The Email Parser Guide: How to Automatically Copy Data From Your Emails*. URL: <https://zapier.com/blog/email-parser-guide/> (visited on 30th Jan. 2020).
- [82] mailparser. *mailparser.io is more than just a software*. URL: <https://mailparser.io/> (visited on 30th Jan. 2020).
- [83] Carlos Martinez. *Filter incoming emails, parse data and write to Excel file*. URL: <https://www.emailparser.com/e/basic-examples/filter-incoming-emails-parse-data-write-excel-file> (visited on 30th Jan. 2020).
- [84] Christopher Olah. *Understanding LSTM Networks*. URL: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> (visited on 25th Oct. 2020).
- [85] Ian Peter. *The history of email*. URL: <http://www.nethistory.info/History%20of%20the%20Internet/email.html> (visited on 7th Jan. 2020).
- [86] Yzmo. *Email*. URL: <https://en.wikipedia.org/wiki/File:Email.svg> (visited on 11th Jan. 2020).

GLOSSARY

ACID Atomicity, Consistency, Isolation, Durability. [41](#)

email Electronic mail. [1](#), [4-7](#), [9-11](#), [25-29](#), [31-35](#), [37](#), [42](#), [43](#), [45-47](#), [49-52](#), [62](#)

GiB GibiByte. [48](#), [49](#)

MB Megabyte. [40](#)

XLNet State-of-the-art cross-lingual understanding through self-supervision. [23](#), [24](#), [39](#)

ACRONYMS

- AI** Artificial Intelligence. 11
- ANN** Artificial Neural Network. 17
- API** Application Programming Interface. 29, 35, 38, 40, 41
- ASCII** American Standard Code for Information Interchange. 10
- AWS** Amazon Web Services. 27–29, 41
- BEAM** Erlang Virtual Machine. 30, 43
- BERT** Bidirectional Encoder Representations from Transformers. 10, 20–24, 26, 37, 38
- BPE** Byte Pair Encoding. 22
- CBOW** Continuous Bag of Words. 14, 15
- CEC** Constant Error Carousel. 18, 19
- CLDR** Unicode Common Locale Data Repository. 40, 46
- DSL** Domain-specific Language. 41
- ERP** Enterprise resource planning. 27
- GC** Garbage Collection. 48
- GloVe** Global Vectors for Word Representation. 15, 16
- GLUE** General Language Understanding Evaluation. 22
- HF** Hessian-free. 18
- HTML** Hypertext Markup Language. 33, 34, 43
- HTTP** Hypertext Transfer Protocol. 1, 27, 29, 40, 41
- IR** Information Retrieval. 11
- IT** Information Technology. 50
- JSON** JavaScript Object Notation. 27, 40, 43, 44
- LSTM** Long Short-Term Memory network. 10, 18, 19
- MIME** Multipurpose Internet Mail Extensions. 10, 32
- ML** Machine Learning. 37
- MLM** Masked Language Model. 21–23, 38
- MT** Machine Translation. 11
- MUA** Mail User Agent. 10
- NER** Named-entity recognition. 10, 11, 13, 17, 21, 23, 24, 26, 29, 37, 39, 43, 44, 48, 50, 52

- NIF** Native Implemented Function. 43
- NLP** Natural Language Processing. 2, 6, 7, 11, 12, 17, 19–21, 29, 38, 51
- OS** Operating System. 43
- PDF** Portable Document Format. 1, 7, 27, 45, 46
- POS** Part-of-speech. 11, 13
- RAM** Random Access Memory. 44, 49
- REST** Representational State Transfer. 35
- RFC** Request for Comments. 10, 25, 26
- RNN** Recurrent Neural Network. 10, 17–19
- SaaS** Software as a Service. 5
- SDK** Software development kit. 6
- SES** Amazon Simple Email Service. 29
- SGD** Stochastic Gradient Descent. 15
- SMTP** Simple Mail Transfer Protocol. 25, 26, 28–30, 42, 43
- SQL** Structured Query Language. 31
- STDIN** Standard Input. 43
- STDOUT** Standard Output. 43
- SVM** Support-vector machine. 12
- TLM** Translation Language Modeling. 22, 23
- UTF-8** Unicode Transformation Format –8-bit. 33, 34, 43
- vCPU** Virtual Central Processing Unit. 49
- WWW** World Wide Web. 5
- XLM** Cross-lingual Language Model. 10, 22–24, 26, 38



SUPPORT MATERIAL

Exchanged emails in a conversation	Occurrence	Percentage
1	2945	70.29%
2	80	1.91%
3	280	6.68%
4	10	0.24%
5	57	1.36%
6	8	0.19%
7	18	0.43%
8	4	0.10%
9	13	0.31%
10	2	0.05%
11	12	0.29%
12	4	0.10%
13	2	0.05%
14	1	0.02%
15	3	0.07%
16	2	0.05%
18	2	0.05%
19	2	0.05%
20	2	0.05%
21	2	0.05%
22	3	0.07%
23	1	0.02%
29	2	0.05%
40	1	0.02%

Table 3: Exchanged [emails](#) in a conversation and it's corresponding occurrence accompanied with the occurrence percent value.

```
1 {
2   "explain": true,
3   "query": {
4     "bool": {
5       "must": {
6         "bool": {
7           "should": [
8             {
9               "dis_max": {
10                "queries": [
11                  {
12                    "match": {
13                      "internal_name": {
14                        "analyzer": "standard",
15                        "boost": 10,
16                        "operator": "and",
17                        "query": "Máscaras MarcaA"
18                      }
19                    }
20                  },
21                  {
22                    "match": {
23                      "internal_name": {
24                        "analyzer": "standard",
25                        "boost": 1,
26                        "fuzziness": 2,
27                        "fuzzy_transpositions": true,
28                        "max_expansions": 20,
29                        "operator": "and",
30                        "prefix_length": 0,
31                        "query": "Máscaras MarcaA"
32                      }
33                    }
34                  }
35                ]
36              }
37            },
38            {
39              "dis_max": {
40                "queries": [
```

```
41     {
42       "match": {
43         "sku": {
44           "analyzer": "keyword",
45           "boost": 10,
46           "operator": "and",
47           "query": "Máscaras MarcaA"
48         }
49       }
50     }
51   ]
52 }
53 }
54 ]
55 }
56 }
57 }
58 },
59 "size": 1,
60 "timeout": "15s"
61 }
```

Listing 12: Elasticsearch query example for text *Máscaras MarcaA*.