Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Inteligencia Artificial

Trabajo Fin de Máster

COpenMed Spanish Medical Corpus: A Large Medical Resource for Entity Normalization

Autor(a): Ana Sanmartín Domenech
Tutor: Igor Boguslavsky
Tutor Externo: Carlos Óscar S. Sorzano

Madrid, Julio 2021

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Inteligencia Artificial

Título: COpenMed Spanish Medical Corpus: A Large Medical Resource for Entity Normalization

Julio 2021

Autora:	Ana Sanmartín Domenech
Tutor:	Igor Boguslavsky Validation and Business Applications Group ETSI Informáticos Universidad Politécnica de Madrid
Tutor Externo:	Carlos Óscar S. Sorzano Natl. Center of Biotechnology (CSIC)

Resumen

La mayoría de la información biomédica se encuentra presente como texto libre y existe un aumento constante de la creación de los documentos relacionados al área médica. El problema principal trata de que el texto libre mencionado se encuentra disponible pero en diferentes fuentes, puesto que no está unificado en un único corpus. Si bien es cierto que la lengua inglesa es el idioma preferido para escribir esos documentos, y existen varios corpus consolidando cierta información, terceros idiomas (como por ejemplo, español) que presentan una amplia información médica, aún necesitan desarrollar un corpus.

Las tecnologías emergentes de Aprendizaje Automático crean y procesan diferentes métodos constantemente y aunque las técnicas matemáticas complejas son relevantes, los corpus de textos son la base del estudio del lenguaje natural.

En ésta tesis de Máster, presentamos un nuevo corpus médico en español denominado **COpenMed Corpus**. El listado original de las páginas web (URLs) empleadas para construir el corpus alcanza la cantidad de 19.571 recursos relacionados con un total de 9.099 entidades. Una función especial de relleno de recursos relacionados a una misma identidad también se ha añadido: la función de relleno se emplea para establecer un valor N referente al mínimo número de referencias por entidad. Para la construcción del corpus se ha empleado $N = 10$ alcanzando un total de **90.990 páginas web** descargadas. Para ésta descarga, se ha desarrollado un método que hace uso de técnicas *Webscraping* para descargar y procesar las páginas web obtenidas del listado.

Como preparativos para un posible futuro trabajo y futuros pasos de uso del corpus creado, se ha pre-entrando dos modelos diferentes de Procesamiento Natural del Lenguaje. Los modelos empleados han sido RoBERTa y XLNet, ambos pre-entrenados con *COpenMed corpus*.

Abstract

The majority of biomedical information is present as free text and there is a constant augment of documents created in a daily basis for this medical field. The main problem is that all of the free text is available but located at different sources, since it is not unified in a single corpus. Although English is the preferred language for writing documents, and has several corpus consolidating the information, another languages with vast amount of information has yet to develop a large corpus.

Emerging *Machine Learning* technologies are appealing and different methods and processes are tested in a constant basis. Even though these complex mathematical techniques are relevant, **text corpora** are the basis of natural language studying.

In this Master thesis we present a new large Spanish medical corpus: **COpenMed Corpus**. Original pool of URLs employed for constructing the corpus reaches the amount of 19.571 resources related to a total of 9.099 entities. A padding method for resources referred to a same entity was also added, the padding is employed by establishing a N value for minimum number of resources per entity. We employed $N = 10$ reaching a total of **90.990 URLs** scraped. For this goal, we develop a method that employs *Webscraping* techniques for downloading and processing a substantial amount of URLs from a given pool of URLs.

As preparations for further steps and future work, pre-training of two different Natural Language Processing models was also done in this Master thesis. Models employed are RoBERTa and XLNet, both of them pre-trained with COpenMed corpus.

Keywords: Medicine Corpus, Corpus Linguistic, Spanish Language Model.

Contents

Chapter 1

Introduction

1.1 Motivation

What is the relation between different illnesses/symptoms? Is a question that may be answered by medical hypotheses. In reality, how inherent relations between terms and words in the medical field is yet to be thoroughly studied. In the last two decades, medical research has evolved from manual discovery of medical term relationship to automatised processes by different emerging techniques such as machine learning. Although this field has been rapidly progressing and different and more complex processes (e.g. neural networks, deep learning, etc.) has gained popularity, there is still to create an extensive medical corpora.

Abundant techniques such as language engineering, human-speech language recognizer models or text analyzers have been improving. Results not only improve day by day but also become more efficient. Although, even if the previous statement is true, no matter what technique is employed, and no matter what technique is developed, all of them depend on the data extracted and engaged in the training process. The corpus is a fundamental part of the mechanism and without it Artificial Intelligence results can not be leveraged.

This Project was created to tackle the need for a useful tool for researchers, scholars and students that allows them to extract, analyze and compare medical information across a wide range of valid medical sources.

1.2 Objectives

In order to enhance current research on medical issues and symptom classification, a solution may be building a language model for easier and faster results. One of the main scientific complications behind this line of research is that the classification and extraction of relations is done by hand which is a tedious and can be a subjective task.

As a starting point towards the goal of creating a Spanish language model for multi-tasking (e.g. Question and Answer, Disease and Symptom classification ...), creating an extensive medical corpora with different topics can be the baseline for employing different modern techniques.

This Master's Thesis aims to (1) download and process the content of different web pages from a pool of URLs, (2) create a medical related topics corpus with the URLs scraped and (3) create two different spanish medical topics trained models. These general goals are translated into the following technical objectives:

- Create a method for automatic *webscraping* of any URL and employ its techniques for downloading the text.
- Make a cleaning process pipeline for the downloaded text and store it.
- Analyze the Natural Language Processing models to be employed and explain step by step the training process.

All processed and stored data was provided previously by the Association COpenMed, the excel file contains a pool of URLs. Information was displayed in an excel file under a Creative Commons CC-BY-SA 4.0 license available at `https://sites.google.com/view/copenmed/descargas`.

1.3 Thesis structure

Chapter 2 presents the state of the art of the COpenMed Project, Spanish Medical Corpora and explains the Machine Learning Models that will be employed in the present project.

Chapter 3 describes the materials and methods used in this Thesis. It thoroughly details how the work pipeline was structured and programmed as well as all its functionalities. It also presents the webscraping process employed, the cleaning texts method and the final training process of the models.

Chapter 4 shows the results and a brief discussion.

Chapter 5 concludes this work and presents possible future research lines. Finally, a detailed referenced literature is provided at the end of the document.

Chapter 2

State of the Art

2.1 COpenMed Association

COpenMed is an nonprofit association with a main objective of offering quality biomedical information. The association origin comes from an idea of creating an automatic reasoner that aids for obtaining a list of possible illnesses given some compatible observed symptoms. COpenMed's work is mainly produced by volunteers and students from different universities such as CEU San Pablo, International University of Valencia, Polytechnic university of Madrid and more.

Information offered is structured in a knowledge graph and internal organization can be understood in the following way:

- **Activity** is composed of subgroups called entities. There are 20 different activities present in the graph: Activity. anatomy, cause, condition, disease, function, groupOfDiseases, groupOfSubstances, groupOfTests, groupOfTreatments, instrument/Device, pathogen, physiological feature, population, risk, specialty, substance, symptom, test, testResult, treatment.
- **Entity** is the subgroup of activity. It can be understood as a principal concept of an specific medical term. Last version from July 2021 has 9099 identified entities.
- **Resource** is each search done or source to complete information about each entity. Las version from July 2021 has 19571 identifies resources.

This structure is also present in an Excel file with a pool of URLs containing the different resources.

2.2 Spanish Medical Corpora

We can distinguish corpus from data set, being the first concept a representative sample with relevant text of a topic and with general purpose, and the second concept as a sample of the corpus with restricted area and annotations involved with a research question.

In the following segments we will explain both of the concepts as well as the most popular datasets/corpus, the available medical datasets/corpus and finally the ones

that are in Spanish.

2.2.1 Non-medical and Medical Datasets comparison

There are a lot of data sets for specific topics, still, the most popular and extensive are not related to the medical area. We have extracted the relevant information such as the name, the language and general purpose as well as the extension. They will be shown below:

- **RACE** is an *english language* dataset [20] for benchmarking evaluation of methods in reading comprehension. The contents include *28,000 texts* and *100,000 questions* generated by human experts for evaluating the students' ability in understanding and reasoning.
- **SQuAD** (*Stanford Question Answering Dataset*) is an *english language* dataset [35] for Question and Answer task. It was created for reading comprehension and it consists of over *100,000 questions* followed by the answer, which is a Wikipedia article passage.
- **IMDB** *(Large Movie Review Dataset)* is an *english language* dataset [24] for the task of binary sentiment classification. It contains a set of 25,000 highly differentiated movie reviews for training, and 25,000 for testing (total of *50,000 reviews*).
- **Yelp** dataset [3] is an *english language* subset of businesses, reviews, and user data from the webpage with its same name. It is mainly employed to teach students or to learn and test different *Natural Language Processing* models. It is one of the largest datasets available containing *over 6 million reviews*.
- **DBpedia** is a community [4] that extracts structured information from Wikipedia allowing sophisticated queries. According to the 2016-04 release, it is composed of *6.0 million* entities. 5.2 million of them are classified in a consistent ontology, being the highest amount related to persons (1.5 million) and the lowest to diseases (*5 thousand*).
- **Amazon Customer Reviews** consist of the contribution of millions of customers by expressing their opinions and description of their experiences regarding purchased products. It contains *over a hundred million reviews*.
- **GLUE** [45] (*General Language Understanding Evaluation*) is an *english language* benchmark consisting of *nine* diverse *Natural Language Understanding* (*NLU*) tasks, which are a collection of different datasets for training specific tasks.

Although we have presented the most popular data sets used, we can observe that the majority of them are related to other expertise areas. The only one that shows medical related topics is DBpedia [4], but it presents a fairly low amount of medical information compared to its highest entity (5 thousand versus 1.5 million).

In order to know if the proposed extension of COpenMED corpus is enough, we have to make another analysis concerning medical knowledge existing corpora. Following the same schema as before, we will extracted the relevant information such as the name, the language and general purpose as well as the extension:

- **GECCO** (*German Corona Consensus Dataset*) is an *English language* uniform

dataset [39] that uses international and health IT standards to establish interoperability of COVID-19 data. It was created to provide comparability between researches and projects of different institutions. It is composed of *81 data elements* with *281 response options* (demography, medical history, symptoms, therapy, medications or laboratory values).

- **National Library of Medicine Database** (*NLM*) is an *image database* [41] for correctly identifying pills. This image database consists of *2,000* high quality reference images and *3,000* lower quality consumer images.
- **IgG4-Related Disease Dataset** [14] was created since on most studies only a few patients with a particular organ manifestation were analysed. The study was conducted on *235 patients* diagnosed in 8 general hospitals in the same medical district.
- **Personalized medicine dataset** presented by Kadi *et al.* [15] was created for the conception of a medical decision-making model. Based on different illnesses included and analyzed, database information reaches up to *500 patients*.
- **fastMRI dataset** [47] is a large-scale collection of images of raw MR measurements and clinical MR, that can be used for training and evaluation of machine-learning approaches to MR image reconstruction. It is a freely-accessible dataset composed by more than *1,500 fully sampled knee MRIs* and *6,970 fully sampled brain MRIs*. This dataset is only accessible for internal research or educational purposes.
- **Taiwan's National Health Insurance Dataset** [7] is a study aimed to analyze and evaluate the frequency and patterns of Chinese Herbal Medicine used in treating osteoarthritis. The database is composed of *22,520,776 outpatients*.
- **The Utrecht dataset** [18] is composed of pulmonary function reference data collected from underage children. It is contains different measurements such as measurements of interrupter resistance with *877 instances*, spirometry with *1042 instances*, body plethysmography with *723 instances* and carbon monoxide diffusion/helium dilution with *543 instances*.

We can see that non large-medical database resources presents Spanish as the language. The largest database is the *Taiwan's National Health Insurance Dataset* [7]. It is relevant to pinpoint that many of the medical databases are prepared for image recognition.

Regarding to all of the datasets explained before, we can observe that the largest and most used are not related to the medical field. We can also underline that all of them are in English language. About medical data sets, a big amount work on medical images or related to information extracted from different patient measurements or for specific illness diagnose or treatment.

2.2.2 Medical Corpus

As presented by Aguilar *et al.* in 2014 [1], we will consider **corpus** as the articulate disposition of different documents with an investigation-related finality. These documents shall be *denaturalized* from its previous structure and *restructured* in a

new form that can be used as an input to a system employed to test a proposed hypotheses.

We will now present the medical corpus found. Let us remember that the corpus to be created must be in Spanish language, although we will show different languages corpus just to be aware of the different languages found, the extension and if they are specific for a task. We will briefly compare the different existing corpus and explain in what topic they revolve:

- The **Quaero French medical corpus** [29] as it name implies, it is a *French language corpus*. Authors report the development of the corpus in French annotated at the entity and concept level. They encompass a total of *103,056 words* with a total of *26,409 entity annotations mapped to 5,797 unique UMLS* concepts.
- **ChiMed** (*Chinese Medical Corpus for Question Answering*) [40] is corpus in *Chinese language* for the challenging task in NLP of QandA. Authors have collected a large-scale Chinese medical information leading to approximately *17.6 million QA pages*. The project has 145 thousand physicians for answering questions in the forum.
- **UKRMED** is a collection and processing project of a medical corpus [8] in *Ukrainian language*. This corpus contains a variety of medical text originated from clinical protocols, blogs or Wikipedia). The UKRainian MEDicine text corpus, combines the following amount of medical texts:
  - Complex texts (from clinical protocols) = *26,730 sentences*.
  - Simple texts (from medicine forums) = *25,395 sentences*.
  - Moderate texts (from Wikipedia) = *27,081 sentences*.
- **Medical Concept Normalization** (*MCN*) is an *English language* comprehensive corpus [23] that links different formal definitions to the same concept in standardized vocabulary. The resulting corpus consists of *10,919 concepts*.
- **emrQA** is a Large Corpus *in English* [30] for Question Answering on Electronic Medical Records. The resulting corpus has *1 million questions* and *400,000+ question-answer pairs*.
- A **corpus-based syntactic study of medical research article titles** was created by Wang *et al.* in 2007 [42]. It analyzes the relationship between *English titles* of medical research articles and their impact. A total of *417 titles* were studied structurally.
- **CLEAR** is a Corpus for Medical French Availability [44]. The corpus is described for the *French medical language*. It contains texts from three sources: encyclopedia, drug leaflets and scientific summaries. The manual processed subset contains *663 pairs* with parallel sentences.
- **COPOS** (*Corpus Of Patient Opinions in Spanish*) [2] is a *Sentiment Analysis* centered corpus in *Spanish language*. The corpus is composed of *743 reviews*, each review contains information about the patient, the medical entity and a numerical value of the opinion expressed,
- **MEDDOCAN** (*Automatic De-Identification of Medical Texts in Spanish*) [26] is a corpus composed of clinical cases. It contains *1,000 clinical case studies*, which

are divided into a train set (500 clinical cases), development set (250 clinical cases) and the test set (250 clinical cases).

- **IULA Spanish Clinical Record Corpus** [25] is a *Spanish language* corpus with *3,194 sentences* extracted from clinical records. The corpus was conceived as a resource to support clinical text-mining systems, but it is also a useful resource for other Natural Language Processing systems handling clinical texts: automatic encoding of clinical records, diagnosis support, term extraction, among others, as well as for the study of clinical texts.

In this section we have presented the most recent and complete medical corpus available. It is exciting to see that different groups awareness of the lack of corpora in other languages. We can observe that the biggest corpus is ChiMed [40], which is reasonable, since due to the large chinese population, higher amount of data will be available. Regarding with Spanish language, we can find COPOS [2], MEDDOCAN [26] and IULA [25]. After analyzing the contents and the aim of these corpus, we can observe that none of them present the same goal as COpenMed, since one of them is extracted from reviews (for Sentiment Analysis) and the other two work over clinical cases. We can then state that COpenMed corpus would be similar to MCN corpus [23] (*Medical Concept Normalization*) since it extracts information from different resources in order to link its contents to the same concept (entity).

Our proposed a corpus will be composed of 9099 different entities and 19.571 resources. Meaning that the final corpus will be formed by 19.571 documents. This amount already surpasses existing available corpora except for *ChiMed* [40] and *MCN* [23]. This corpus will have the possibility of *padding* the amount of resources to a default amount N for each entity. We employed $N = 10$, so COpenMed corpus first version will be composed of a possible **90.990 documents** surpassing any *spanish medical corpus* created to the date.

2.3 Machine Learning Models

After presenting the different existing corpora, since another assignment of this Project is to pre-train two different language models with the created corpus, let us now present the context of Machine Learning Models. Afterwards a brief presentation of the two different models chosen will also be done.

The dominant approach of Machine Learning systems is to employ large datasets of training examples and supervise its learning process to demonstrate the behavior for an individual desired task [19], after training the system, the behavior is tested on independent and identically distributed (IID) datasets. This process works fairly well for narrow experts, but the problem is that these systems need a prepared dataset since slight changes in the data may tweak results due to the high sensitivity and task specification [36, 16].

In order to resolve these task specification and sensitivity flaws, new methods were built. For example, word vectors were used as inputs [17, 9], then contextual representations of recurrent networks [10, 31], and recent work suggests that task-specific architectures may not be necessary anymore.

We can observe in Figure 2.1 the schema proposed for a Single Task Model. It is specific for Natural Language Processing since the input is Text. Since the input text

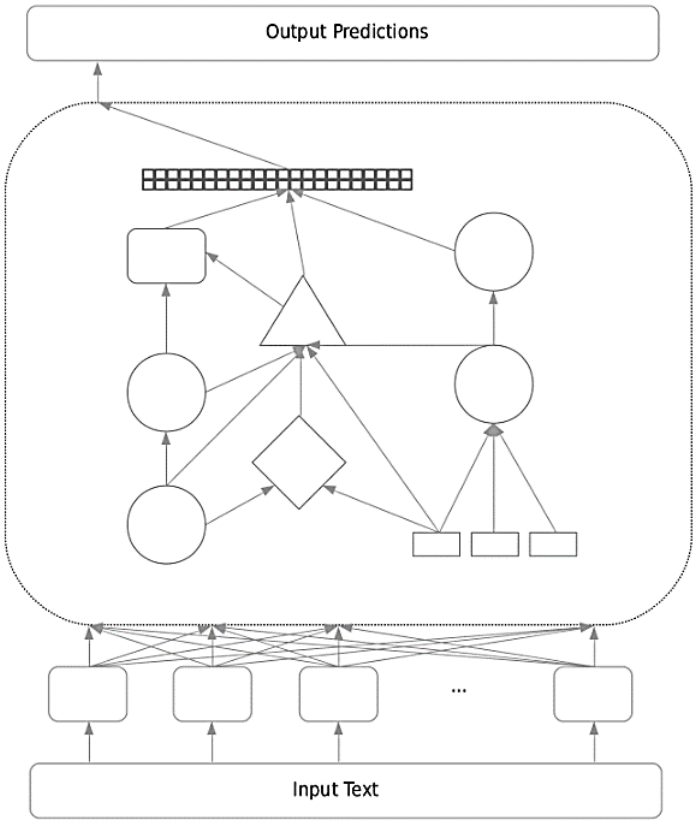

Figure 2.1: Single Task Model example schema. Figure adapted from GLUE original paper [43].

is a unique and individual input, it can be understood that the model is created to provide results for a specific task.

Although the best performing models had specialized architectures the just mentioned sensitivity problem decreases the performing value to non acceptable values. To avoid this problems, progress heads for robust systems. These systems present architectures compatible with different tasks, but requires training and measuring performance on a large of tasks.

We can consider this movement towards robust systems as a new movement of preference towards more general systems (can perform many tasks). The problem appears when considering whether in practice, optimization of the model with reach convergence or not. It is confirmed that large language models are able to perform multitask learning but learning process is much slower. Another problem is the labeling process. Labeling manually the same or several datasets for different tasks is an arduous assignment. Since current Machine Learning Systems need a vast amount of examples for a well generalization. This suggests that multitask training may need as much training examples as a single-task model for each task. This fact motivated for the creation of benchmarks for multitask learning by employing pre-training or/and supervised fine-tuning. Nowadays, GLUE [43] and decaNLP [28] benchmarks appeared and are the most used to aid this training and measuring process.

Unlike single task models that trains end-to-end a specific task, multitask models extensively train on a generalized language understanding task. This is based on an extensive generalized pre-training of the model, this process helps to obtain *generalized "understanding"* that proceeds with model adjustment to approach a specific task.

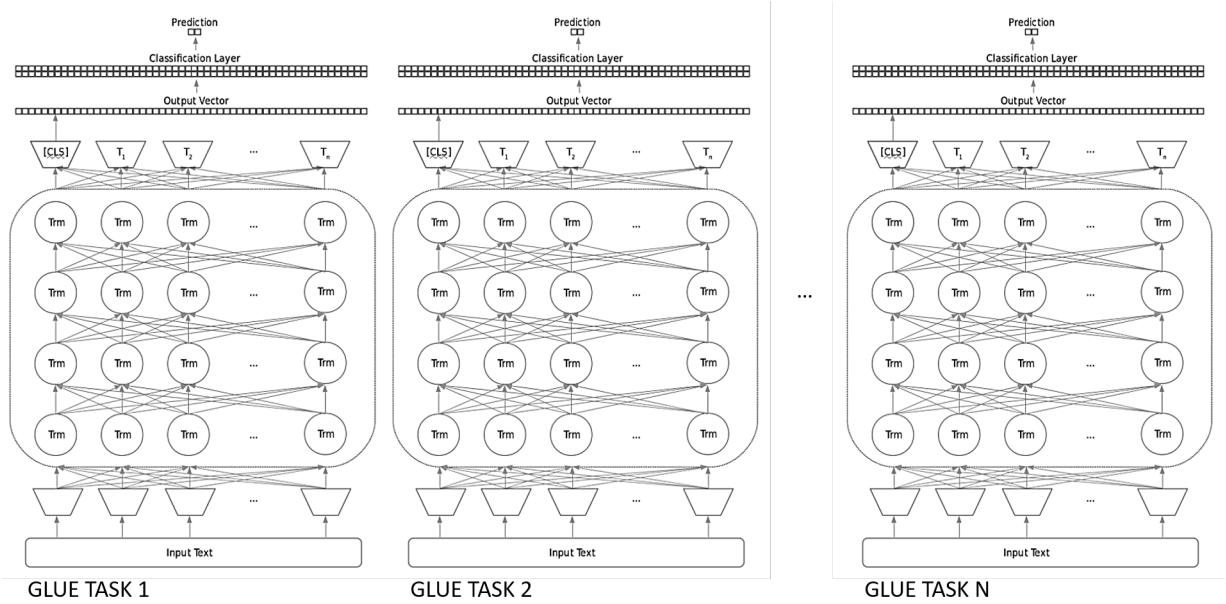

Figure 2.2: GLUE Benchmark schema. It is composed of 9 tasks, so $N = 9$. Figure adapted from GLUE original paper [43].

We can observe in Figure 2.2 the general structure of GLUE. GLUE [43] is a collection of tasks for evaluating a model. It is composed of 9 tasks and the resulting average score of the 9 tasks is the model's final score. GLUE is structured in a way that the architecture of the model is not relevant, but the only requirement is that input and output prediction are accommodated to each GLUE task. We will briefly show the different tasks so that the reader can see what kind of tasks are the most relevant ones.

- CoLA dataset: Trains for recognition if a sentence grammar is correct.
- SST-2 dataset: Trains Sentiment Analysis.
- MRPC dataset: Trains if sentence B is paraphrase of sentence A.
- STS-B dataset: Trains for recognition of the similarity between two sentences.
- QQP dataset: Trains for recognition of the similarity of two questions.
- MNLI-mm dataset: Trains for the recognition if a sentence B entails or contradicts sentence A.
- QNLI dataset: Trains for a Question and Answer system.
- RTE dataset: Trains for the entailment of a sentence.
- WNLI dataset: Trains for the correct pronoun replacement.

After explaining briefly how multitask models are structured, we are now prepared to present multilingual models. The multilingual model approach is quite similar to multitask approach. The model is trained on a text corpus composed of many languages. Since amount of documents imbalance is not impossible, it is preferred that in case a lack of texts in certain language is present, the imbalance is regulated with the input of extra articles. This occurs for certain small languages. This way, small languages are oversampled and large languages are undersampled.

Language model pre-training has been proven to improve Natural Language Processing tasks [10, 34]. Some of these tasks are present in the GLUE benchmark, and were just mentioned before for example, paraphrasing [13] and question answering [38, 35].

There are two existing strategies for applying pre-trained language models to downstream tasks: feature-based and fine-tuning.

- Feature-based approach uses task-specific architectures that include the pre-trained representations as additional features. One example is ELMo [31]
- Fine-tuning approach needs minimum task-specific parameters, and is trained simply by fine-tuning all pretrained parameters. One example is the GPT (*Generative Pre-trained Transformer*) [33].

The major limitation of these two approaches is the unidirectionality of standard language models. This restrictions can be very harmful in the fine-tuning process for token-level tasks (e.g. question answering).

To improve the fine-tuning approach, *Bidirectional Encoder Representations from Transformers* (BERT) model [12] was proposed.

Previous work on model probing has shown that the induced language model is able to encode syntactic and named entity information [31]. Auto-regressive (AR) language modeling and AutoEncoding (AE) have been the two most successful pretraining objectives. Inspired by AR language modeling, XLNet model [46] was also proposed.

These two models mentioned will be the ones employed in this Master thesis. Although we will not use the original flavour of BERT, it will explained since their architectures and functionalities are almost the same.

2.3.1 BERT: Bidirectional Encoder Representations from Transformers

For resolving the unidirectionality constraint, BERT model employs a "*masked language model*" (*MLM*) pre-training objective. This masked language model, randomly masks some of the tokens from the input and it tries to predict the original vocabulary based on the context.

By using *MLM*, the model must take the text surrounding (left and right) the masked word, allowing to pre-train a deep bidirectional Transformer. When BERT was published, it achieved state-of-the-art performance outperforming many task-specific architectures.

We can observe the two steps of BERT in Figure 2.3. The two steps in their framework are pre-training and fine-tuning.

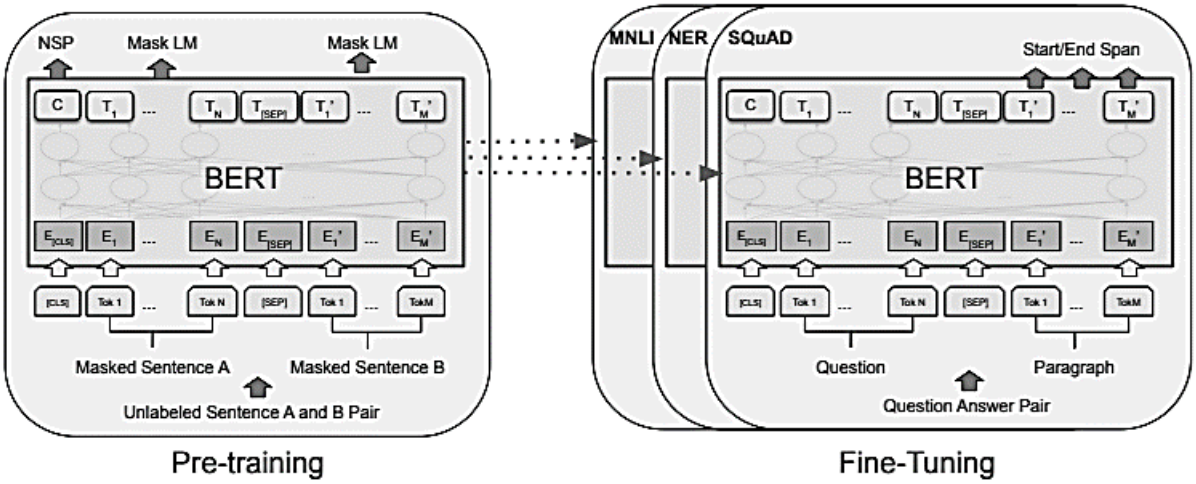

Figure 2.3: BERT structure. We can observe Pre-training step on the left and Fine-tuning step on the right. Figure adapted from BERT original paper [12].

During the pre-training process, the model is trained on unlabeled data over different pre-training tasks. Afterwards, the model will go through a fine-tuning process. BERT model is firstly initialized with the parameters from the pre-training step and those parameters will be fine-tuned with labeled data.

BERT model also presents a **multilingual model release** (M-BERT). It is trained on pooled data from 104 languages. M-BERT has been shown realistic results generalizing across the 104 languages [32], but monolingual BERT models easily outperform M-BERT. For example the French BERT model [27], the German BERT model [37] and the Spanish BERT [6].

Another smaller multilingual BERT mode composed of 15 languages [21] augmented the results obtained from M-BERT.

Aside from multilingual models, some studies focus on pre-training a BERT model on particular subdomains such as BioBERT [22] or SciBERT [5] which are trained with biomedical publications and scientific texts.

2.3.2 XLNet: Generalized Autoregressive Pretraining for Language Understanding

XLNet is proposed to leverage by extracting the best of both Auto-regressive (AR) language modeling and Auto-encoding (AE). Given a text sequence $x = [x_1, ...x_T]$ Auroregressive maximizes the likelihood under the forward autoregressive factorization:

$$\log p_\theta(x) = \sum_{t=1}^{T} \log p_\theta(x_t|x_{<t}) = \sum_{t=1}^{T} \log \frac{\exp\left(h_\theta(x_{1:t-1})^T e(x_t)\right)}{\sum_{x'} \exp((h_\theta(x_{1:t-1})^T e(x')))}$$

where

- $h_\theta(x_{1:t-1})$ is a context representation produced by neural models

- $e(x)$ is the embedding of x

Auto-regressive language model is only trained to encode a uni-directional context and is not effective for modeling bidirectional contexts (on the contrary to BERT). In comparison, pre-training aims to reconstruct original data by replacing a certain portion of tokens by a special symbol [MASK]. The model is then trained to recover the original tokens from the corrupted version.

XLNet has three different improvement over BERT:

- Maximizes the expected log likelihood of a sequence. Also, it implements a permutation operation, allowing tokens to use contextual information from all positions.
- XLNet does not suffer from the pretrain-finetune discrepancy.
- XLNet is inspired by the Auto-regressive language and is based on Transformer-XL [11], improving the performance.

In the following section we will show the comparison presented by *Yang et al.* for XLNET and BERT and BERT-like model. This comparison will be useful to decide if these two models will be implemented.

2.3.3 BERT vs XLNet: Empirical performing results

Under comparable experiment environment, XLNet [46] outperforms BERT [12]. Including GLUE, BERT and XLNet were evaluated in language understanding tasks. Let us present Table 2.1 results of evaluating both models of different tasks.

Model	MNLI	QNLI	QQP	RTE	SST-2	MRPC	CoLA	STS-B	WNLI
ST: BERT	86.6	92.3	91.3	70.4	93.2	88.0	60.6	90.0	-
ST: RoBERTa	90.2	94.7	92.2	**86.6**	96.4	90.9	68.0	92.4	-
ST: XLNet	**90.8**	**94.9**	**92.3**	85.9	**97.0**	**90.8**	**69.0**	**92.5**	-
MT: RoBERTa	90.8	98.9	90.2	88.2	96.7	92.3	67.8	92.2	89.0
MT: XLNet	**90.9**	**99.0**	**90.4**	**88.5**	**97.1**	**92.9**	**70.2**	**93.0**	**92.5**

Table 2.1: Empirical performing accuracy results of BERT, RoBERTa and XLNet. ST = Single Task, MT = Multitask. Table adapted from *Yang et al.* [46].

After analysing Table 2.1 we can observe that XLNET's accuracy results exceeds BERT's. Also, RoBERTa's accuracy values seem better and higher than BERT's. Hence, we have decided to employ RoBERTa for one of the models instead of BERT due to the values shown on the table above.

Chapter 3

Materials and Methods

Let us remember three objectives that we want to achieve in this Master's Thesis:

- Create a method for automatic *webscraping* of any URL and employ its techniques for downloading the text.
- Make a cleaning process pipeline for the downloaded text and store it to create the corpus.
- Analyze the Natural Language Processing models to be employed and explain step by step the training process.

In the following sections, point 3.2 will cover objectives 1 and 2 by explaining the process in which the raw corpus is created. The pipeline employed and a brief explanation of the different input methods will also be presented in this step. Additionally, it also explains the installation and usage process in order to use the program.

Section 3.3 will cover objective 3 by not only presenting the two different pretraining methods for **Natural Language Processing** (*NLP*) but also a brief analysis on how to create the pre-training of the model.

Before explaining all the steps, let us present figure 3.1 as the proposed pipeline in this Master's thesis.

Figure 3.1 clearly differentiates and shows the separation established for the different tasks to be done in this project. Three different tasks were found when structuring the whole program: downloading and creating the corpus, creation of the input files for the model and pre-training of the models. These tasks are separated into different steps (from 1 to 3) and explained in detail in the following sections.

As a note to the reader, all scripts were written in Python version 3.8.10 on a dual boot *Windows 10* and *Linux 20.04.2 LTS*. GPU computations were run on a 16GB memory *GeForce GTX 1660 SUPER*. Some calculations were run on Google Collab since training time were exponentially high.

3.1 Folders internal structure

This section will present the **different scripts** created for the correct functioning of the project and a **brief explanation** of their contents. This will allow the reader to

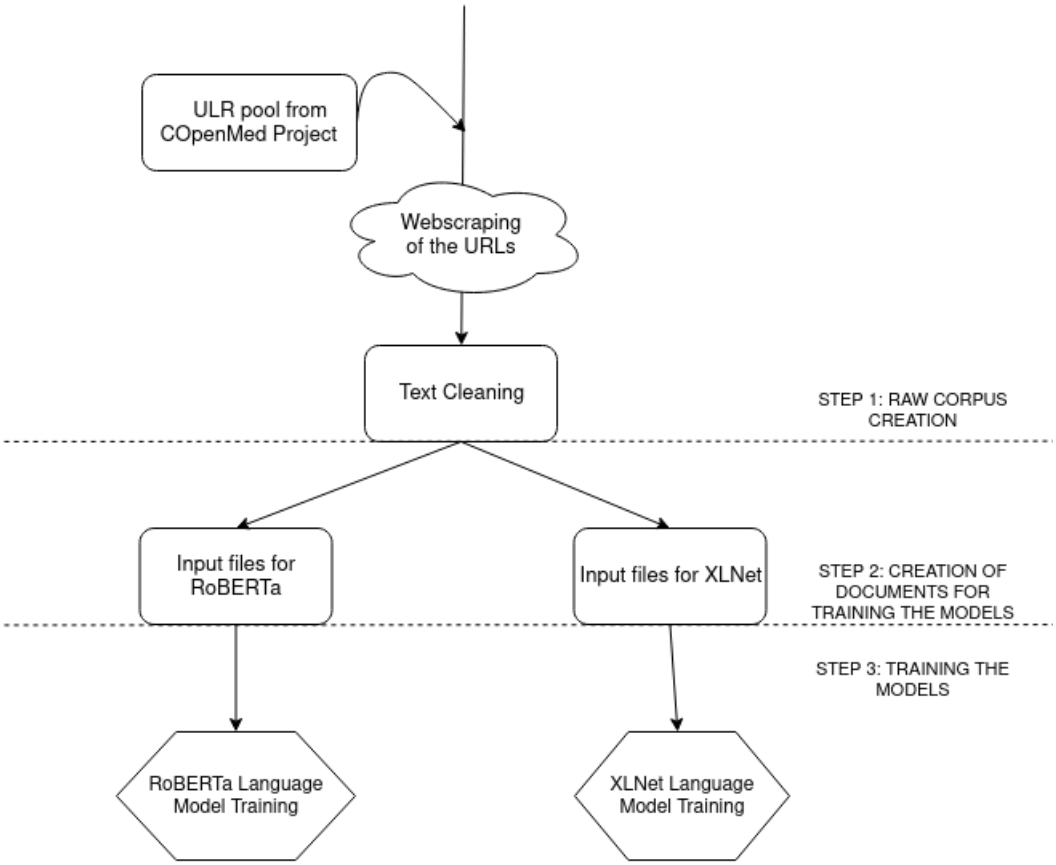

Figure 3.1: General schema for the project. We can observe that the proposed pipeline shows 3 different steps, which are the same explained above on the objectives.

know how the code was structured and separated. If the reader wants to know the specific usage of a script, it will be explained below in the following sections.

This project is separated into **3 different folders**: webscraping, dir and training. Let us now explain in detail the files contained.

webscraping folder

- **url_searcher.py**: python file that contains the code involved in *reading an excel file* with specific structure. It extracts the entities names and the URLs to be searched.
- **searcher.py**: python file that contains the code involved in *searching a word in the search engine* (www.google.es) and storing the URLs related to the entity into a plain text file (url_list.txt).
- **downloader.py**: python file that contains the code involved in *downloading the content of the URLs* which can be read from a plain text file (url_list.txt) It processes the text and selects the most relevant paragraphs to be stored in a new plain text file (content.txt from dir folder).
- **check_resource.py**: python file that contains the code involved in *checking if an entity folder has been created*, if the URL to be searched has been searched

o not and modifies the status of each search in order to optimize time search if future additions to the database are done.

- **preprocessing.py**: python file that contains the code involved in *preprocessing the raw text* (*NLP* cleaning text methods) in order to further computations for extraction of relevant text.
- **url_list.txt**: plain text file that contains in each line the *URL to be searched* followed by the entity name. Both terms separated by a blank space.
- **temporal.pdf**: file for *temporal storage of a pdf* file downloaded from and URL. It will be deleted after it is converted into a plain text file.

dir folder

- **entity folder**: entities are integer numbers (id's) that have a medical term mapped. The relation between both of then can be found in the URLs excel (for more information about the excel see section 3.2.1). *Each folder will be named with its entity id.*
  - **status.txt**: plain text file contains for each entity, the *status of the URLs* (searched, not searched or blacklist).
  - **search folder**: each folder's name goes from 1 to N. It represents the number of URLs searched for a single entity.
    - **content.txt**: plain text file that contains the plain cleaned final text.
    - **url.txt**: plain text file that contains the URL from where the text has been scraped.

training folder

- **spm_input.py**: python file that contains the code involved in *writing in the same plain text file* the 20 % of the whole data set. It extracts resources 1 and 2 of each entity into the same document with each sentence in a row.
- **glob_input.py**: python file that contains the code involved in *writing in the same plain text file* the 100 % of the whole data set. It extracts all the resources of each entity into the same document with each sentence in a row and each resource separated by an empty row.
- **spm_input.txt**: plain text file that contains the 20% of the complete data set.
- **glob.input.txt**: plain text file that contains the 100% of the complete data set.
  - **RoBERTa folder**: folder containing tokenizer and training model for RoBERTa model.
    - **merges.txt**: plain text file that contains merged tokenized sub-string for RoBERTa model.
    - **vocab.json**: json file which contains the indices of the tokenized sub-strings for RoBERTa model.
      - **runs**: folder where different runs of RoBERTa model and their results are stored.

- **XLNet folder**: folder containing tokenizer and training model for XLNet model.
  - **merges.txt**: plain text file that contains merged tokenized sub-string for XLNet model.
  - **vocab.json**: json file which contains the indices of the tokenized sub-strings for XLNet model.
    - **runs**: folder where different runs of XLNet model and their results are stored.

3.2 Step 1: Raw corpus creation

In this section we will thoroughly explain the process of creation of the corpus. As we can observe in Figure 3.2 the program reads and obtains a list of URLs and entities related to them. Afterwards it automatically searches those URLs on a google search engine.

After that, the downloaded text will be filtered according to 14 different filters created for the most numerous URLs in the excel where the URLs will be read. The following step is a typical simple relevance ranking problem in which we store the test with relevance above a certain threshold.

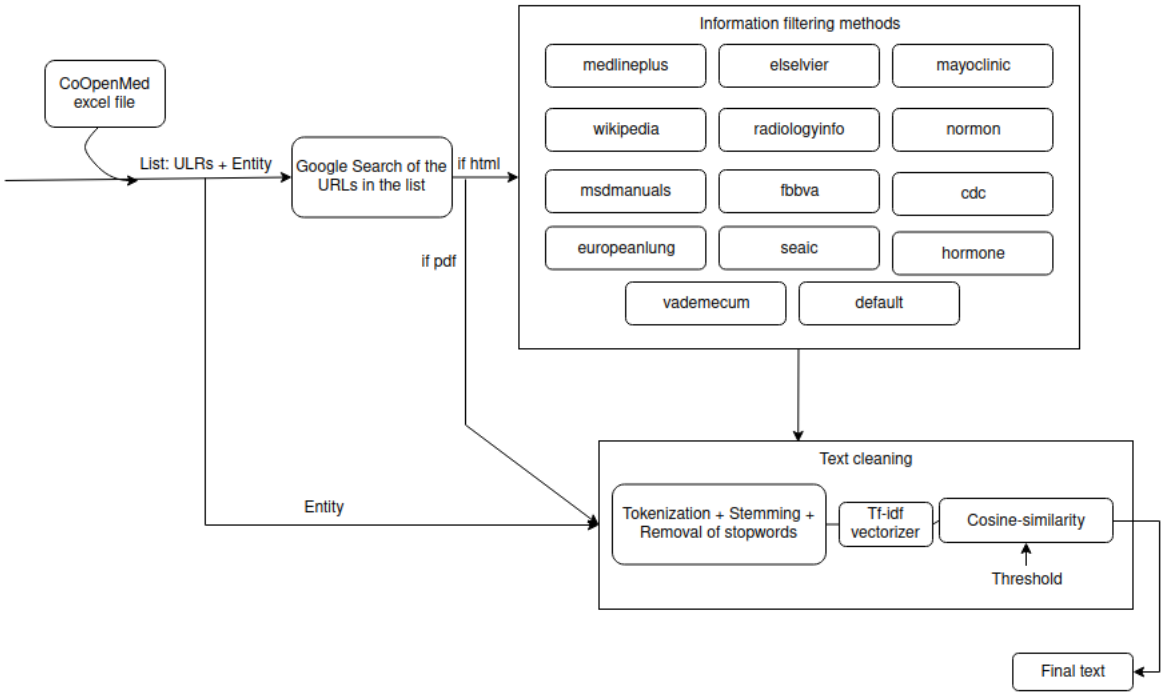

Figure 3.2: Raw corpus creation schema. We can observe the workflows: the intput URL is an html or the input is a PDF file.

3.2.1 Expected input file

As mentioned before, there are **two** different type of input files:

- **Batch search** by inputting an excel file with the URLs to be downloaded.
- **Individual search** by inputting a single entity name and downloading *N URLs* related to that entity (where *N* is a modifiable value).

Despite having different inputs, both batch search and individual search dump the input data into a plain text file (*url_list.txt*). We can see below some psudo code lines about the functionality: the program checks the status of the URL, and depending on the value read $(0, 1, -1)$ the decision of ignoring or adding the URL to the search list will be taken.

```
for line in xml:
    read URL, entity_name

    if entity_name_file exists:
        read URL_status in status.txt:
            if URL_status == 0:
                write in url_list.txt:
                    URL + " " + entity_name
            if URL_status == 1:
                pass
            if URL_status == -1:
                pass
            if URL_status not in status.txt:
                append URL + " " URL_status = 0
                write in url_list.txt:
                    URL + " " + entity_name
    else:
        write in url_list.txt:
            URL + " " + entity_name

after webscraping process is over
    delete url_list.txt
```

For further information on how to use the commands to differentiate *batch search* and *individual search* modalities please see section 3.2.6.

3.2.1.1 Online vs Offline

The program supports online and offline usage.

- **Online mode** can be used on batch search and on individual search. When using online mode with batch search, only the URL will be provided and its contents will be retrieved. When using the individual search either entity name or directly an URL can be provided to the program.
- **Offline mode** can only be used from batch search. The content of the *URL* must be the partial path (starting from the working directory). The file must be a PDF file and the program will convert it into a plain text file with the most relevant information according to the entity.

We can observe in Table 3.1 an example of the data needed for *batch search* (either online and offline modes).

Following the schema in Table 3.1 and reading the pseudo code below, we can understand that **online mode and offline mode will be automatically differentiated**.

Neccesary data in the excel input file		
Entity Id	Entity Name	Resource URL
1	$Entity\_Name_1$	URL_1
2	$Entity\_Name_2$	URL_2
...	$Entity\_Name_{...}$	$URL_{...}$
N	$Entity\_Name_N$	URL_N

Table 3.1: Necessary information in the excel input. The excel at least needs to have Entity Id, Entity name and the Resource URL in the information.

Since the code will read the document by analyzing line by line and since the analysis of each line is independent from the one above or below, both modes can be used in the same document since the program will know what mode to rise.

```
for each line in excel:
    URL = read resource_URL
    if URL starts_with https:
        do: online mode
    else:
        if check_pdf_path_exists URL:
            do: offline mode
        else:
            raise URL_status = 0
```

3.2.1.2 *html* vs PDF

Some *PDF*s are published online and, although, they present URLs for the content extraction, text from them can't be obtained as the rest of the URLs since they do not present a *html* structure. The **distinction point is the URL extension**: if it ends with .pdf it will be analysed and extracted as a PDF file and if not, as html. Both of these methods will be explained below:

For ***html* scraping**, a general and simple scraping process was implemented (downloading all the *div* content), but since results were lousy, specific filters for each webpage were created. To create a good filter, at least 5 different pages from the same domain were checked by hand. After that a filter that accepts the highest amount of relevant information and rejects the highest amount of irrelevant information such as menu options and references present in the majority of the URLs was created.

We can see in Table 3.2 the different filters employed. For the 13 webpages that have specific filters, the data extracted is fairly clean and the majority of non relevant lines are rejected. Filter number 14 is the most defective since it is a standard filter that accepts almost all text.

14 different filters employed for *webscraping*		
Filter Id	**Filtered webpage**	**Location of relevant information**
1	Medlineplus	 Relevant text Relevant text

2	Wikipedia	Relevant text <li class = "" id = ""> Relevant text

3	Mayoclinic	
		   Relevant text
		  <ul class = "" id = ""> Relevant text
		
		<article id = "main-content">
		   Relevant text
		  <ul class = "" id = ""> Relevant text
		</article>

4	Elsevier	<section id = "texto-completo"> Relevant text </section>
5	Radiologyinfo	 Relevant text Relevant text
6	Normon	 Relevant text Relevant text
7	Msdmanuals	 Relevant text

8	Fbbva	
		    Relevant text
		   Relevant text
		   Relevant text
		     Relevant text

9	Cdc	 Relevant text Relevant text

10	Europeanlung	
		   Relevant text
		   - Relevant text
		
		   Relevant text
		   - Relevant text

11	Seaic	 Relevant text Relevant text
12	Hormone	 Relevant text Relevant text
13	Vademecum	 Relevant text Relevant text
14	Default	 Relevant text Relevant text

Table 3.2: Filters employed for an optimized cleaning of the URLs content. Since the structure of the data is so diverse from each webpage, a special cleaning was indispensable.

For **PDF scraping**, a third party python package called ***tika*** was employed. This package allows the user to extract all the text from a PDF file as plain text. Although it is unquestionable easier than *html* scraping process, it loads all the text, including index, references, editorial notes... which are not relevant for the program.

3.2.2 Webscraping

Extracting the content of multiple webpages can be an arduous task since the majority of their structures present different complexity. Also, if the scraping process does not follow a *human-like* behaviour, such as spending some time in a webpage (humans do not *click* on different webpages in less than a second, or send multiple requests within milliseconds) it can rise suspicions on the scraped webpage. Different strategies can be employed so as not to raise suspicions of the information extracting process. The created program offers the user two different options in order to successfully scrape different webpages:

- Scraping process **with credential**.
- Scraping process **without credential**.

3.2.2.1 With credential

The program is compatible with some external applications (e.g. Scraping bee) that offer certain services such as header rotation, IP rotation, VPN... These functionalities allows the user to freely scrape any webpage without the blacklist concern. Secure

scraping applications provides the user with a set of values that have to be included in the code when downloading the information.

Instead of writing the values directly in the code, which is an extremely non-safe approach, the program only needs the user to include a file named *credentials.txt* inside the *webscraping* directory. The code automatically detects the file and will employ the credentials if present in the folder. This file must include the provided values as a *python dictionary-like* structure as shown below.

```
{"api_key": "the_api_key","url": "the_url_from_the_api",}
```

3.2.2.2 Without credential

Considering that the majority of safe scraping applications charges an usage fee, some users may not be willing to pay the cost (e.g. individual users). For this reason, the program is also prepared for this situation.

Different scenarios were tested:

- **Header rotation**: The computer and search engine version and model were extracted randomly from a pool for each search. Although it seemed like a feasible solution, each webpage presents different headers (different orders and requirements) and they rejected the majority of the requests since the modified header *seemed suspicious* to them.
- **IP rotation**: Another possible solution was to make requests with different IPs. The IPs employed for the test were the ones freely available at:...BUSCAR PAGINA. This solution was not viable since those IPs are publicy located, and thus are constantly used for testing, making the majority of them already in the blacklist of a lot of webpages.
- **Waiting some time**: The easiest but not optimal solution was implemented. A human takes around half a minute to do a superficial reading of the text and consider if it is interesting or relevant. This characteristic was the one implemented in the code.

3.2.2.3 *Webscraping* dependencies

For **Webscraping**, two third party python packages called ***requests*** and ***BeautifulSoup*** were employed. These packages allows conjunct usage for extracting all the text from an URL.

- **Requests v. 2.25.1**: Standard library for making HTTP requests in Python. Allowing the user to customize the request's headers and data and inspect data from the requests and the responses.
- **BeautifulSoup v. 4.9.3**: Library for making easier the scraping process. It sits atop an HTML or XML parser, providing functions for iterating, searching, and modifying the parse tree.

3.2.3 Natural Language Texts Processing method

A large amount of text is scraped from each *webpage* and it is clear that non relevant text will also be downloaded. To solve this problem, a simple **Information Retrieval**

Ranking method can be implemented.

We can observe in Figure 3.3 a basic schema for a Ranking Information Retrieval System. In this process, a ranking algorithm will compute the query against all documents and extract a relevance parameter. This relevance parameter expresses the importance of each document regarding the query inserted. The higher the value, the higher the relevance, and viceversa. We will plan on our information filtering system on this Information Retrieval System schema.

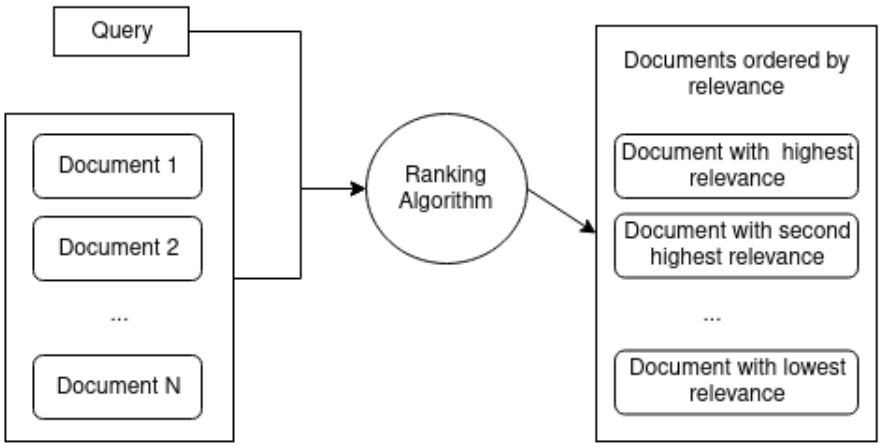

Figure 3.3: Basic schema for a Ranking Information Retrieval System.

Figure 3.4 represents the specific structure to be employed in this project. As explained before it is an adjusted version of Figure 3.3. For this specific case, the *query* is equivalent to the *entity name*, and each document is equivalent to a paragraph in a document. After the application of the Ranking Algorithm, instead of ranking the paragraphs, we will filter the paragraphs by a **dynamic threshold**.

For the Ranking Algorithm we will employ basic Natural Language Processing (*NLP*) Operations. We can observe the cleaning process in Figure 3.5. The raw text will go through six different operations:

- Tokenization
- Stemming(*)
- Conversion to lower case.
- Removal of stop words.
- Removal of words with 2 or less characters.
- Removal of non alphanumeric.

(*) The goal of both **stemming** and **lemmatization** is to reduce inflectional forms to a common base form. Although both operations are feasible and valid, lemmalization's results are normally better for retrieving the base or dictionary form of a word, since stemming normally chops off the end of words. For this program, we have chosen to employ stemming over lemmalization due to stemming faster processing time.

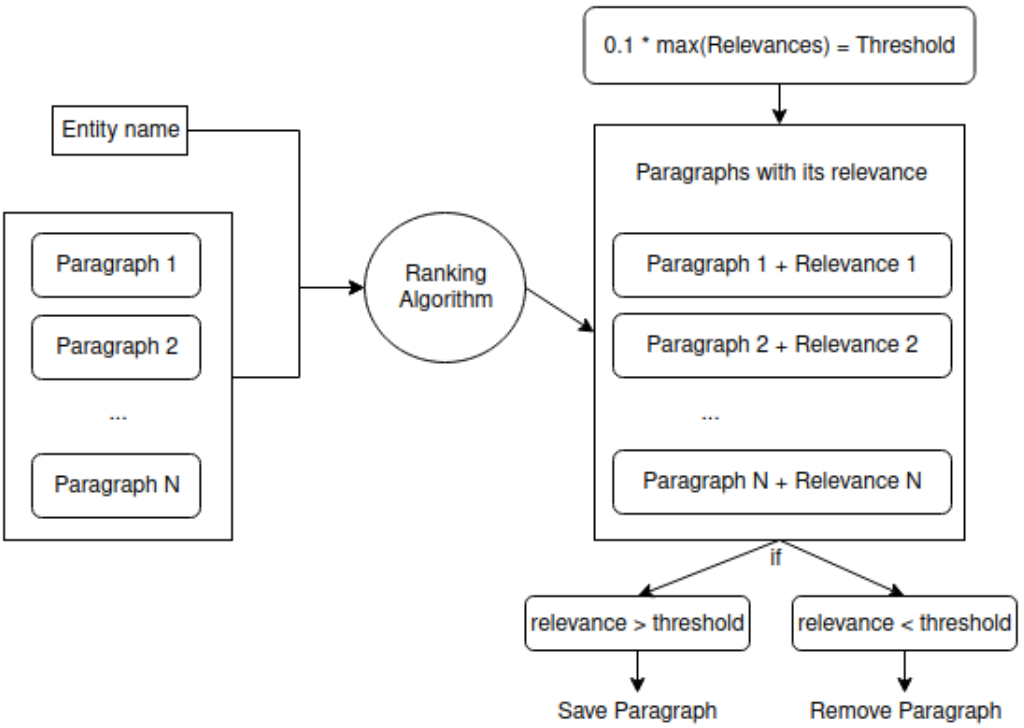

Figure 3.4: Adapted version of a schema for a Ranking Information Retrieval System for our particular problem.

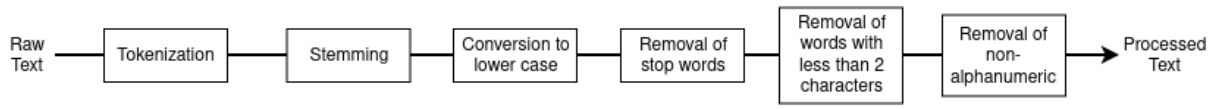

Figure 3.5: Natural Language Processing (*NLP*) Operations applied to the text.

3.2.4 Ranking Algorithm

Since we need to compute an easy and fast algorithm, **Cosine similarity** was implemented. Cosine similarity measures the **similarity between two vectors** by obtaining the cosine of the angle between the two vectors.

It is one of the most widely used and it is present in multiple applications such as finding similar documents in *NLP*, information retrieval or detecting plagiarism.

For this step, the third party python package employed is ***scikit-learn***. Since we have documents instead of vectors (and cosine similarity functions parameters are a pair of vectors). Please see Figure 3.6 for the schema. We firstly employ *TfidfVectorizer* function to convert the collection of paragraphs and the entity name to two different matrices of TF-IDF features. After obtaining the features, we can employ the *cosine_similarity* function.

We can observe in Figure 3.7 the cosine vector operation applied to a scraped text.

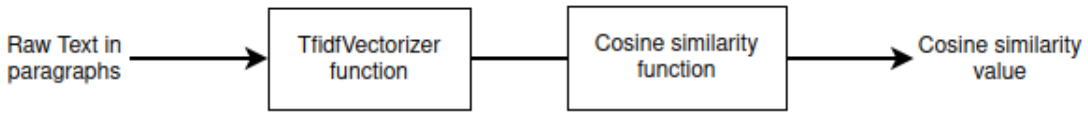

Figure 3.6: Ranking Algorithm employed.

Each numerical value refers to the cosine between entity name and paragraph. Threshold value is set to $0.1 * max(value)$, if the cosine value is lower that than number, the paragraph will not be stored. Let us pinpoint that the saved text will the original text (raw) and this processing process is only employed to know which parts are the most relevant ones.

```
[0.         0.         0.         0.         0.         0.38256554
 0.3054493  0.17073416 0.27498756 0.21127905 0.48907622 0.30846196
 0.12551456 0.         0.54835186 0.36939636 0.62318372 0.45968586
 0.17011135 0.22752128 0.         0.         0.14652296 0.05865017
 0.14743671 0.34069227 0.33327403 0.2325699  0.27307415 0.54116458
 0.23619547 0.         0.2121565  0.22301056 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.15298202 0.11118614 0.1673141
 0.         0.         0.         0.         0.         0.
 0.10060583]
Threshold employed for this document is 0.062318372059473694
```

Figure 3.7: Output of applying cosine similarity operation on entity name vector and paragraphs-vectors in a document.

3.2.5 Output files

The program stores in two different plain text files the final raw content and the URL from where the information was scraped. In Figure 3.8 we can clearly observe the followed schema for the output files: each entity is separated in a different folder and each resource corresponding to one entity is stored inside such folder. The functionality of knowing what URLs have been searched successfully and unsuccessfully has also been added by the presence of status.txt file inside each entity (1 and -1, respectively).

This process of storing the content of each URL and the URL searched is done after each web page is scraped. After all of the URLs in the list of URLs to be searched have been processed, the program will go through the list again, but instead of searching for the contents again, it searches if the URL inside the list is present or not in a status.txt file. If it isn't present, this means that there was an error when reading the URL or the search process terminated unexpectedly, meaning that the web page contents were not downloaded. Thus, the URL shall be added to the corresponding status.txt with a status $= 0$.

The program created needs to receive the working directory as an input argument.

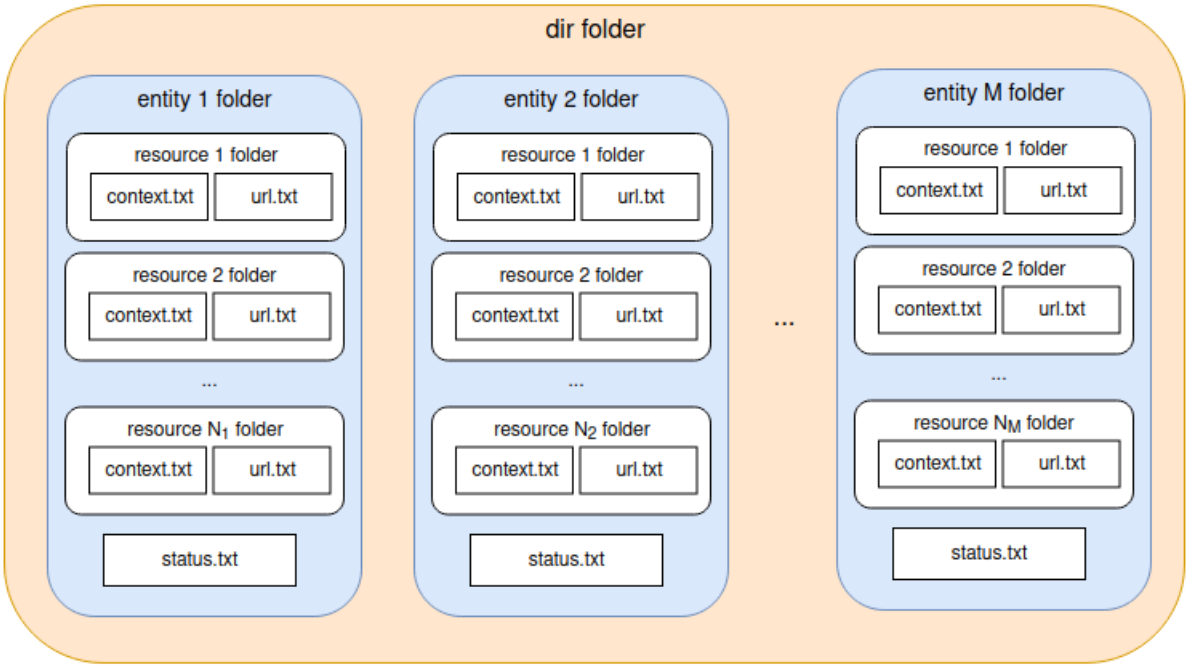

Figure 3.8: Schema of the directory distribution.

This input parameter needs to be where the directory *dir* is located or the user wants it to be located. The code automatically detects if the *dir* folder exists or not as well as entity and resource folders. In order to have an homogeneous folder style, entity folder name will be the *entity id* instead of the entity name, as well as the resource folder name, it will be a number from 1 to N where N is the number of folders inside entity directory. We can see below the pseudocode employed for a resource storing process.

```
# If the dir folder does not exist, create it
if not dir_folder.exists:
    mkdir dir_folder

# Enter the dir folder
enter dir_folder

# If the entity folder does not exist, create it
if not entity_folder.exists:
    mkdir entity_folder

#Enter the entity folder
enter entity_folder

# Create the resource folder
mkdir resource_folder_(enumeration+1)

# Create the content and the URL texts files
write content.txt
write url.txt

# Finally change the status to 1, or add the URL with status 1 if successfull
change url_status in status.txt
```

3.2.6 User manual

In this section, we will explain how to use correctly the program for both batch search and individual search modalities.

Individual search modality usage is the following:

```
searcher.py [-h] [-w WORD] [-u URL] [-n N] [-s SEARCH] [-o OUTPUT]
```

- The argument *-w* is the **word or short phrase** to be queried in the search engine. If a short phrase is inserted, spaces must be exchanged for underscores (_). The program does not differentiate between lower caps and upper caps (a *lower()* function is implemented) but the presence of *accent marks* influences on the retrieved results. Parameter *w* has **no predefined value**.
- The argument *-u* is the **URL** to be scraped. This will fetch a single search, so no *N argument* is needed, and it is present, it will be ignored. Parameter *u* has **no predefined value**.
- The argument *-n* is the **number of URLs** to be retrieved for a single search. The program is structured so that if the proposed URL by the search engine has been stored before, it will ignore it and process the next proposed URL. By default the value of *n* is **10**.
- The argument *-s* is the **search engine** to be employed. Although the user can choose a preferred one, it is recommended to not no modify this parameter. By default the value of *s* is **google search engine**.
- The argument *-o* is the **output directory** to be employed. It must be the global path to where the corpus will be downloaded. By default the value of *o* is the **working directory**.

The user can't use *WORD argument* and *URL argument* at the same time. Even if both parameters are sent to the program, it will prioritize the URL search and will ignore the number present at the *N argument* (since the retrieval of a specific URL will only receive one single search).

Batch search modality usage is the following:

```
url_searcher.py [-h] [-d DOCUMENT] [-m MIN_SEARCHES] [-c
↪  CREDENTIALS] [-o OUTPUT]
```

- The argument *-d* is the **excel file** with entity names and URLs to be searched. It is a mandatory parameter and does not have by default values.
- The argument *-m* is the **minimum number of retrievals** done by entity. This means that after adding all the URLs in the excel document to the plain text file, the program will pad the number of resources present in each entity if it has less than *m* (see Figure 3.9). By default the value of m is **10**.
- The argument *-c* is the global path to the **credential text file** for safe *webscraping*. It is not a mandatory parameter and if the user does not specify it by default the program will search the file in the **working directory**.
- The argument *-o* is the **output directory** to be employed. It must be the global path to where the corpus will be downloaded. By default the value of *o* is the **working directory**.

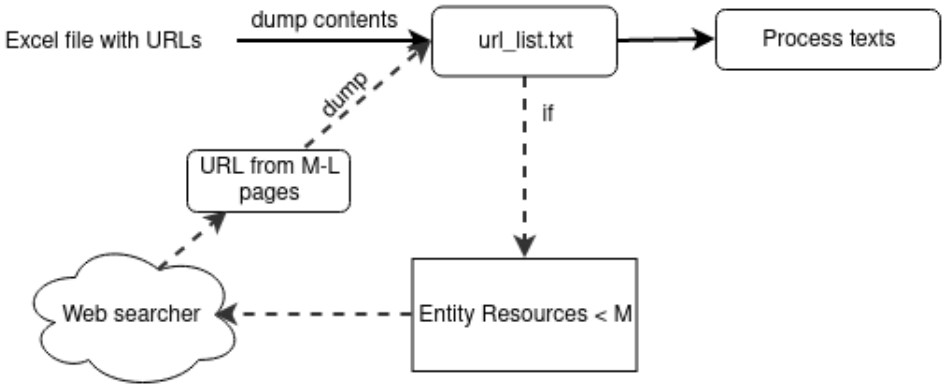

Figure 3.9: Padding process related to url_searcher.py *m* argument. *M* is the number input by the user and *L* is the actual number of resources in an entity.

3.3 Step 2: Creation of documents for pre-training the models

In this section, we will explain the process followed for **pre-training two different models from scratch**. As explained before, since we want the models to be focused on the medical area, **no pre-trained tokenizers** or models will be used. We will focus on building a transformer model built from *Hugging Face's modules*. Usage of the Transformers library features was preferred over training separately directly with the models raw scripts due to the unified pipeline for all the models available in their interface.

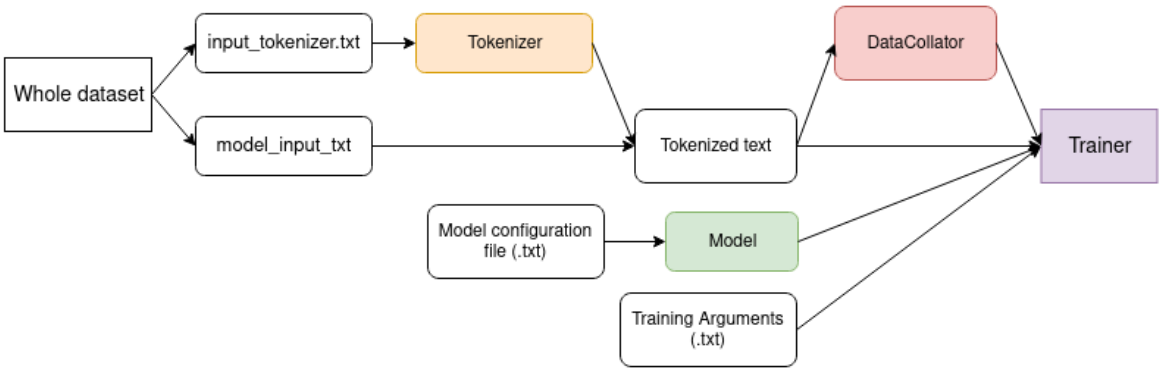

Figure 3.10: General schema employed for the model creation.

Figure 3.10 shows the general schema employed from the whole dataset to the final training process. We will explain separately Tokenizer (yellow box) and Model (green box) steps, since is where RoBERTa and XLNet have their main differences.

3.3.0.1 Loading of the dataset

In the previous section (3.2) we created a cleaned corpus, which we will need to separate for the tokenizer and for the training process.

Since is not ideal that the tokenizer is biased by any medical topic (we understand a medical topic as each entity), pre-training of the tokenizer shall be done over a small corpus with *at least one document from each entity*. For this, we call spm_input.py. The program will read resources 1 and 2 from each entity and write them in a plain text file called spm_input.txt. As shown below, each sentence will be in a separate line without character differentiation between documents.

```
line1 of document1
line2 of document1
line1 of document2
line2 of document2
line3 of document2
...
lineN of documentM
```

For the *training process document,* we will call glob_input.py. The program will read all the resources from all the entities and write them in a plain text file called glob_input.txt. As shown below, each sentence will be in a separate line and a new line character between documents.

```
line1 of document1
line2 of document1

line1 of document2
line2 of document2
line3 of document2
...
lineN of documentM
```

3.3.1 Transformer for RoBERTa

RoBERTa (**A Robustly Optimized BERT Pretraining Approach**) model, as it name indicates, is one of the numerous BERT-like descendant models. This model is based on BERT's architecture and modifies key hyperparameters, removes the next-sentence pretraining objective and optimizes larger mini-batches training and learning rates. Meaning that it has **improved the pretraining process**.

This segment will explain step by step the schema for the pretraining of the model. It will follow the next points:

1. Training a tokenizer from scratch.
2. Initializing a model from scratch.
3. Build of the database.
4. Define of the Data Collator.
5. Initializing the trainer.
6. Pretraining the model.

Let us now explain step by step the building of our RoBERTa model:

3.3.1.1 Training a tokenizer from scratch

In contrast to the original BERT, which uses **Word Piece tokenization**, RoBERTa uses **Byte Pair Encoding** (*BPE*), which means that it descends to a byte-level tokenization. The paremeters employed for training the tokenizer will be the following:

- files = spm_input.txt
- vocab_size = 52000
- min_frequency = 2
- special_tokens = [ ]
  - <s> = start token.
  - <pad> = padding token.
  - </s> = end token.
  - <unk> = unknown token.
  - <mask> = mask token for language modeling.

The code employed is the following:

```
from tokenizers import ByteLevelBPETokenizer

path = '/PATH/TO/SPM/INPUT/spm_input.txt'

# Initialize Byte Pain Encoding (BPE) tokenizer
tokenizer = ByteLevelBPETokenizer()

# Train tokenizer with file and parameters
tokenizer.train(files=path, vocab_size=30_522, min_frequency=2,
                special_tokens=['<s>', '<pad>', '</s>', '<unk>', '<mask>'])

# Create model folder if necessary
# os.mkdir('/PATH/TO/MODEL/ModelName')

# Save the created tokenizer (files vocab.json and merges.txt will be created and stored in
    this folder and step)
tokenizer.save_model('/PATH/TO/MODEL/ModelName')
```

3.3.1.2 Initializing a model from scratch

The first step before initializing a model is to define the *parameters for the configuration* of the model. We can observe the parameters chosen and the code below: The parameters employed for training the tokenizer will be the following:

- vocab_size = 52000
- max_position_embeddings = 512
- num_attention_heads = 12
- num_hidden_layers = 6
- type_vocab_size = 1

```
from tokenizers import RobertaTokenizerFast, RobertaConfig

```

```
# Establish the configuration parameters
config = RobertaConfig(
    vocab_size = 52000,
    max_position_embeddings = 512,
    num_attention_heads = 12,
    num_hidden_layers = 6,
    type_vocab_size = 1,
    )

# Initialize a RoBERTa model with the configuration parameters
model = RobertaForMaskedLM(config=config)
```

3.3.1.3 Build of the database

The raw data cannot enter the training process, this means that we have to *encode* the texts with the pre-trained tokenizer we just created. As we explained before, RoBERTa uses Byte Pair Encoding (BPE), which means that it descends to a byte-level tokenization. This tokenizer has been trained to treat spaces like parts of the tokens.

The post-processor will add a start (<s>) and an end token (</s>) as well as other encoding parameters. Below we can observe the code employed for this step:

```
from transformers import RobertaTokenizerFast, LineByLineTextDataset

# Load the tokenizer we created in the previous step
tokenizer = RobertaTokenizerFast.from_pretrained('/PATH/TO/MODEL/ModelName')

#Encode line by line out dataset with the loaded tokenizer
dataset = LineByLineTextDataset(
    tokenizer=tokenizer,
    file_path='/PATH/TO/SPM/INPUT/glob_input.txt',block_size=128
    )
```

3.3.1.4 Define of the Data Collator

Data Collator is an object to for back-propagation, it is created before initializing the trainer and its function is to *assemble samples from the dataset into batches*, creating dictionary-like objects. Since our model is called RobertaForMaskedLM (Masked Language Modeling) we must set mlm property to True.

Below we can observe the code employed for this step:

```
from transformers import DataCollatorForLanguageModeling

# Data Collator with the tokenizer, MLM and masked tokens to 15%
data_collator = DataCollatorForLanguageModeling(
    tokenizer = tokenizer,
    mlm = True,
    mlm_probability = 0.15
    )
```

3.3.1.5 Initializing the trainer

After doing of all the preparations from before, the trainer is now ready to be initialized. We just need to establish the *training arguments*, which will be below, as well as the code:

- output_dir = Path to out model directory
- overwrite_output_dir = True
- num_train_epochs = 2
- per_device_train_batch_size = 64
- save_steps = 10000
- save_total_limit = 2

```
from transformers import Trainer, TrainingArguments

# Establish the training parameters
training_args = TrainingArguments(
        output_dir = '/PATH/TO/MODEL/ModelName',
        overwrite_output_dir = True,
        num_train_epochs = 1,
        per_device_train_batch_size = 64,
        save_steps= 10000,
        save_total_limit = 2,
)

# Trainer needs a model, the arguments, the data collator and the encoded dataset
trainer = Trainer(
        model = model,
        args = training_args,
        data_collator = data_collator,
        train_dataset = dataset,
)
```

To reach this step we had to create and structure a wide amount of methods. Let us remember what we needed: the **Trainer** needs as input the RoBERTa model created in step 3.3.1.2, the **training arguments** we created in this step (3.3.1.5), the **data collator** initialized in step 3.3.1.4 and finally the **encoded dataset** from step 3.3.1.3.

3.3.1.6 Pretraining the model

Finally we are ready for pretraining the model. The trainer is launched with the following line:

```
# Launch the trainer
%%time
trainer.train()

# Save the trainer in the model file
trainer.save_model('/PATH/TO/MODEL/ModelName')
```

The output of this training process shows the batch size per device, the optimization steps, the loss, learning rate and epoch and the employed time.

3.3.2 Transformer for XLNet

XLNet is aunsupervised language representation learning method. It is an extension of the *Transformer-XL* (thus, implementing it as well). It is based on *Transformer-XL* model architecture and employs an autoregressive method to learn bidirectional contexts by maximizing the expected likelihood on the input sequence.

XLNet implementation on this project was decided since under comparable experiments settings, it outranks BERT base in 20 tasks language tasks (e.g. question answering, sentiment analysis, document ranking...)

This segment will explain step by step the schema for the pretraining of the model. It will follow the next points:

1. Training a tokenizer from scratch.
2. Initializing a model from scratch.
3. Build of the database.
4. Define of the Data Collator.
5. Initializing the trainer.
6. Pretraining the model.

Let us now explain step by step the building of our XLNet model:

3.3.2.1 Training a tokenizer from scratch

In contrast to the previously pre-trained RoBERTa model, XLNet employs a tokenizer based on *SentencePiece*. XLNet uses **Unigram**, which, in contrast to *BPE* or *WordPiece*, it initializes its base vocabulary (all pre-tokenized words and most common substrings) to a large number of symbols and progressively trims down to obtain a smaller vocabulary. *Unigram* is normally used in conjunction with *SentencePiece* (we will implement it in a conjunctive way as well).

At each training step, the algorithm computes the possible loss if a symbol was to be removed from the vocabulary and removes the one with the lowest value. The *Unigram algorithm* always keeps the base characters so that any word can be tokenized.

The paramaters employed for training the tokenizer will be the following:

- files = spm_input.txt
- vocab_size = 52000
- special_tokens = [ ]
  - <s> = start token.
  - <pad> = padding token.
  - </s> = end token.
  - <unk> = unknown token.
  - <mask> = mask token for language modeling.

The code employed is the following:

```
from tokenizers import SentencePieceUnigramTokenizer

path = '/PATH/TO/SPM/INPUT/spm_input.txt'

# Initialize SentencePiece's Unigram tokenizer
tokenizer = SentencePieceUnigramTokenizer()

```

```
# Train tokenizer with file and parameters
tokenizer.train(
    files=path,
    vocab_size=52000,
    special_tokens=['<s>', '<pad>', '</s>', '<unk>', '<mask>']
    )

# Create model folder if necessary
# os.mkdir('/PATH/TO/MODEL/ModelName')

# Save the created tokenizer (files vocab.json and merges.txt will be created and stored in
    this folder and step)
tokenizer.save_model('/PATH/TO/MODEL/ModelName')
```

3.3.2.2 Initializing a model from scratch

The first step before initializing a model is to define the *parameters for the configuration* of the model. We can observe the parameters chosen and the code below: The parameters employed for training the tokenizer will be the following:

- vocab_size = 52000
- n_head = 12
- n_layers = 6

```
from tokenizers import XLNetTokenizerFast, XLNetConfig

# Establish the configuration parameters
config = XLNetConfig(
    vocab_size = 52000,
    n_head = 12,
    n_layer = 6,
    )

# Initialize a XLNet model with the configuration parameters
model = XLNetModel(config=config)
```

3.3.2.3 Build of the database

The raw data cannot enter the training process, this means that we have to *encode* the texts with the pre-trained tokenizer we just created. As we explained before, XLNet uses a *SentencePiece* inspired *Unigram* implementation. It is a fairly simple process since the input text is treated as a sequence of Unicode characters. Whitespace is also handled as a normal symbol and is replaced with a meta symbol, **underscore** (U+2581).

The post-processor will add an underscore at each whitespace which will be appended to the word behind. Below we can observe the code employed for this step:

```
from transformers import XLNetTokenizerFast, LineByLineTextDataset

# Load the tokenizer we created in the previous step
tokenizer = XLNetTokenizerFast.from_pretrained('/PATH/TO/MODEL/ModelName')

#Encode line by line out dataset with the loaded tokenizer
dataset = LineByLineTextDataset(
    tokenizer=tokenizer,
    file_path='/PATH/TO/SPM/INPUT/glob_input.txt',block_size=128
    )
```

3.3.2.4 Define of the Data Collator

Data Collator is an object to for back-propagation, it is created before initializing the trainer and its function is to *assemble samples from the dataset into batches*, creating dictionary-like objects. Since our model is called XLNetModel we must set mlm property to False.

Below we can observe the code employed for this step:

```
from transformers import DataCollatorForLanguageModeling

# Data Collator with the tokenizer, MLM set to false and masked tokens to 15%
data_collator = DataCollatorForLanguageModeling(
    tokenizer = tokenizer,
    mlm = False,
    )
```

3.3.2.5 Initializing the trainer

After doing of all the preparations from before, the trainer is now ready to be initialized. We just need to establish the *training arguments*, which will be below, as well as the code:

- output_dir = Path to out model directory
- overwrite_output_dir = True
- num_train_epochs = 2
- per_device_train_batch_size = 64
- save_steps = 10000
- save_total_limit = 2
- learning_rate = 5e-5,

```
from transformers import Trainer, TrainingArguments

# Establish the training parameters
training_args = TrainingArguments(
        output_dir = '/PATH/TO/MODEL/ModelName',
        overwrite_output_dir = True,
        num_train_epochs = 1,
        per_device_train_batch_size = 64,
        save_steps= 10000,
        save_total_limit = 2,
        learning_rate = 5e-5,
)

# Trainer needs a model, the arguments, the data collator and the encoded dataset
trainer = Trainer(
        model = model,
        args = training_args,
        data_collator = data_collator,
        train_dataset = dataset,
)
```

To reach this step we had to create and structure a wide amount of methods. Let us remember what we needed: the **Trainer** needs as input the RoBERTa model created in step 3.3.2.2, the **training arguments** we created in this step (3.3.2.5), the **data collator** initialized in step 3.3.2.4 and finally the **encoded dataset** from step 3.3.2.3.

3.3.2.6 Pretraining the model

Finally we are aready for pretraining the model. The trainer is launched with the following line:

```
# Launch the trainer
%%time
trainer.train()

# Save the trainer in the model file
trainer.save_model('/PATH/TO/MODEL/ModelName')
```

The output of this training process shows the batch size per device, the optimization steps, the loss, learning rate and epoch and the employed time.

3.3.2.7 Extracting model embeddings

Embeddings are useful for tasks such as clustering or semantic textual similarity. The process followed for embedding is the following:

First, we need the model created at the beginning which will be the word embedding model, which will map tokens in a sentence to the output embeddings. The next layer in the model is called *Pooled model*, which, will perform a mean-pooling operation since we need to have a fixed size sentence vector (the mean of all the pool of sentences). After that, by the use of the *SentenceTransformer* function we store the final model with the embedding model and the pool.

```
# Load our model
word_embedding_model = models.Transformer('/PATH/TO/MODEL/ModelName')

# Since there should be a fixed sized sentence vector, we apply a mean pool
pooling_model = models.Pooling(word_embedding_model.get_word_embedding_dimension(),
                               pooling_mode_mean_tokens=True,
                               pooling_mode_cls_token=False,
                               pooling_mode_max_tokens=False)

model = SentenceTransformer(modules=[word_embedding_model, pooling_model])
```

NLIDataReader is employed to load the dataset and generate a dataloader for training the SentenceTransformer model. For train loss any function is available, such as Softmax Classifier,

```
nli_reader = NLIDataReader('/PATH/TO/NLI/DataSet')

train_data = SentencesDataset(nli_reader.get_examples('train.gz'), model=model)
train_dataloader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
train_loss = losses.SoftmaxLoss(model=model, sentence_embedding_dimension=model.
    get_sentence_embedding_dimension(), num_labels=train_num_labels)
```

Finally, the training of the model will be the following:

```
model.fit(train_objectives=[(train_dataloader, train_loss)],
          epochs=num_epochs,
          evaluation_steps=1000,
          warmup_steps=warmup_steps,
          output_path=model_save_path
          )
```

Chapter 4

Results and Discussion

The achievement of our three objectives is assessed in the next sections. We will explain each step and the results obtained at each phase of the process.

Section 4.0.1 will thoroughly show the input and output obtained for the achievement of the first and second objective, section 4.0.2 will explain and show the results of the models results (covering objective 3).

4.0.1 *Webscraping* results

4.0.1.1 Load URLs into URL list text file

The first step the user does, is to download the text from the URLs. The program will automatically load into the URL list text file all the URLs present in the excel file that have to match (that have not been downloaded before). We can see in Figure 4.1 the output of the program. It shows the user the URLs to be inspected, the number of elements in the present excel and the element in which the program is.

```
The url https://www.nhlbi.nih.gov/health-topics/espanol/pruebas-de-la-funcion-pulmonar has no match, adding to text file
Number of elements in excel is: 11950
We are in: 748
The url https://empendium.com/manualmibe/chapter/B34.V.25.4.6. has no match, adding to text file
Number of elements in excel is: 11950
We are in: 753
The url https://mibebeyyo.elmundo.es/bebes/alimentacion/lactancia/ingurgitacion-mamaria has no match, adding to text file
Number of elements in excel is: 11950
We are in: 760
The url https://www.studocu.com/es/document/pontificia-universidad-javeriana/anatomia-radiologica-y-fisiologia/apuntes/2-op
```

Figure 4.1: Output of the program when loading the URLs into the URL list text file.

After that, the program will take each line of the URL list file in order to search for the content. We can see the content of the file called url_list.txt in Figure 4.2. As expected, the content of this file is the URL followed by the entity name. Let us take into account that the entity name contains non ASCII characters (such as accent marks), when proceeding to the cosine similarity step, the entity name to be introduced as the *query* shall also contain the same non ASCII characters for higher numerical value of the cosine.

Let us also explain that inside the file, if the entity name is composed of more than 1 word (which is the norm), it will separated by a blank space. The internal code will

automatically differentiate between the URL and the entity.

```
1 https://medlineplus.gov/spanish/ency/article/000819.htm Alergia farmacológica
2 https://medlineplus.gov/spanish/ency/article/000817.htm Alergia alimentaria
3 https://medlineplus.gov/spanish/ency/article/000033.htm Reacción alérgica por mordeduras y picaduras de insecto
4 https://medlineplus.gov/spanish/druginfo/meds/a682133-es.html Morfina
```

Figure 4.2: url_list.txt content. URL is followed by the entity name.

4.0.1.2 Scraping Process of URLs

The next step will be the scraping process which can be seen in Figure 4.3. At the beginning, the program will inform the user basic information about the scraping process to be done.

- Usage of credential or not. Let us remember that this credential is given by an external program. If a non-working credential is employed the program will be blocked until the credential is modified or the file is removed from the working directory.
- Response HTTP Status Code shows if the downloading process was done successfully or not. If the result is 200, everything has been scraped correctly.
- Encoding of the *webpage* and confidence of the encoding. This is an important step since Spanish language presents characters such as accents, which may have decoding errors if they are not treated correctly.
- Filter employed for the scraping process. This filter reads the beginning of the URL to be scraped and decides which filter to employ.

```
No credential given
Response HTTP Status Code:  200
{'encoding': 'utf-8', 'confidence': 0.99, 'language': ''}
ISO-8859-1
We have stored the content in data
We are cleaning text as Cdc
```

Figure 4.3: Output before the *webscraping* process. The program informs the user if a credential is being used, the response HTTP status code, the encoding type and the filter to be employed.

4.0.1.3 Cleaning of the text and Cosine Similarity Result

Afterwards, the program automatically downloads the texts following the chosen filter instructions. Separates the full text into paragraphs and chooses which paragraph to maintain and which to drop regarding to the cosine similarity result. We can see an example in Figure 4.4. We can see the cosine similarity result (the first line containing 4 numbers). The program takes the maximum value of the cosine similarity and sets the threshold to the 10% of that value. As seen in Figure 4.4 the maximum value is 0.12284648, thus the threshold employed is 0.012284648. By employing this threshold, paragraphs one, two and four will be stored and paragraph three will be dropped since the cosine value is smaller than the threshold.

```
<class 'list'>
[0.1183799  0.12284648 0.         0.08134708]
[0.1183799  0.12284648 0.         0.08134708]
Threshold employed for this document is 0.01228464751709295
```

Figure 4.4: Output for cosine similarity results and the threshold employed for a document.

4.0.1.4 Output of the Process: content.txt and url.txt content

Finally, the text is stored in a context text file (as shown in Figure 4.5). We can observe that the text is written as raw text and special characters are kept.

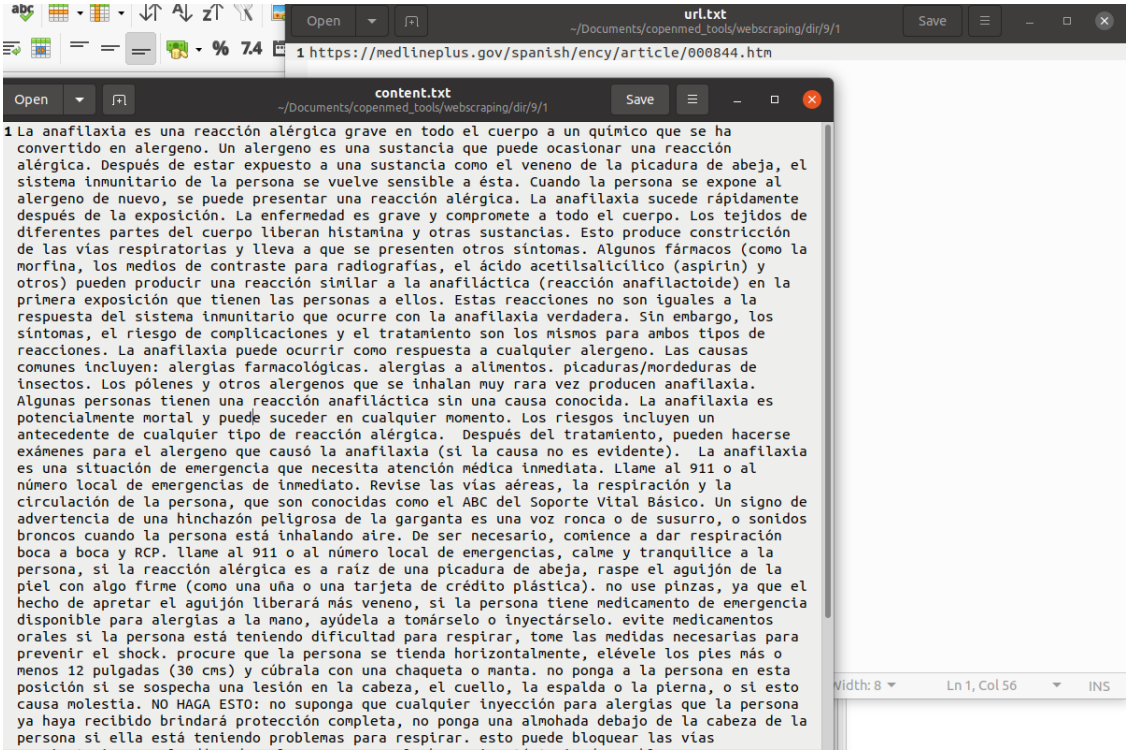

Figure 4.5: Example of the text in content and in url text files.

We just presented the results for the webscraping process, then, in the next section, we will present the results for the training of the models.

4.0.2 Training of the models

We will follow the same fashion as the previous section for presenting the results.

4.0.2.1 GPU requirements

Although a local GPU was employed, processing time was exponential and GPU was always out of memory. We can observe in Figure 4.6 the high RAM GPU from Google Collab employed for calculations.

```
Mon Jul  5 11:12:39 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 465.27       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   37C    P0    26W / 250W |      0MiB / 16280MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 4.6: High RAM GPU from Google Collab. This is shown when running *nvidia-smi*

4.0.2.2 Initialization of the models

After checking and preparing the GPU requirements for training the models, let us show the model created in Figure 4.7. We can observe that it is a BERT (RoBERTa) model with 6 layers and 12 heads.

```
RobertaForMaskedLM(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(52000, 768, padding_idx=1)
      (position_embeddings): Embedding(512, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): RobertaEncoder(
      (layer): ModuleList(
        (0): RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
```

Figure 4.7: Print of the model. We can clearly see that it is a RoBERTa Model

4.0.2.3 Pre-Training Process

The next step is the training process. We can observe in Figure 4.8 the training parameters employed for both training processes. The number of examples, equals the number of sentences employed in the training, which is over 1 million. The batch size was selected to 64 to avoid RAM out of memory errors.

```
***** Running training *****

  Num examples = 1299401
  Num Epochs = 5
  Instantaneous batch size per device = 64
  Total train batch size (w. parallel, distributed & accumulation) = 64
  Gradient Accumulation steps = 1
  Total optimization steps = 23395
```

Figure 4.8: Training parameters for the training process

4.0.2.4 Training times

We can see in Figures 4.9 and 4.10 the time employed by the models for the training process in a high RAM GPU. RoBERTa trained for half the time XLNet needed.

```
Training completed. Do not forget to share your model on huggingface.co/models =)


CPU times: user 1h 12min 33s, sys: 1min 2s, total: 1h 13min 36s
Wall time: 1h 11min 7s
```

Figure 4.9: Training time needed for RoBERTa Model.

```
Training completed. Do not forget to share your model on huggingface.co/models =)


CPU times: user 3h 1min 14s, sys: 2min 36s, total: 3h 3min 50s
Wall time: 2h 57min 36s
```

Figure 4.10: Training time needed for XLNet Model.

For a GeForce GTX 1600 SUPER, the run time for both models exceeded one week of computing.

4.0.2.5 Training Loss

In Figure 4.11 and in Table 4.1 we can observe the Training Loss of the models.

The shape of the learning curve and the training loss curve can be used to diagnose the behavior of a machine learning model. This behavior can suggest the application of some possible changes that may improve performance. There are three dynamics in learning curves:

- Underfit refers to a model that cannot learn the training dataset.
- Overfit refers to a model that has learned the training dataset too well.

- Good fit exists between an overfit and underfit model and is identified by a training and validation loss decreased to a stable point between the two final loss values.

If we take a closer look at Figure 4.11 we can tell clearly that both models show **underfitting**. An underfit model can be identified from the learning curve of the training loss only: RoBERTa model shows a flat line and XLNet presents a decreasing training loss that continues to decrease until the end of training.

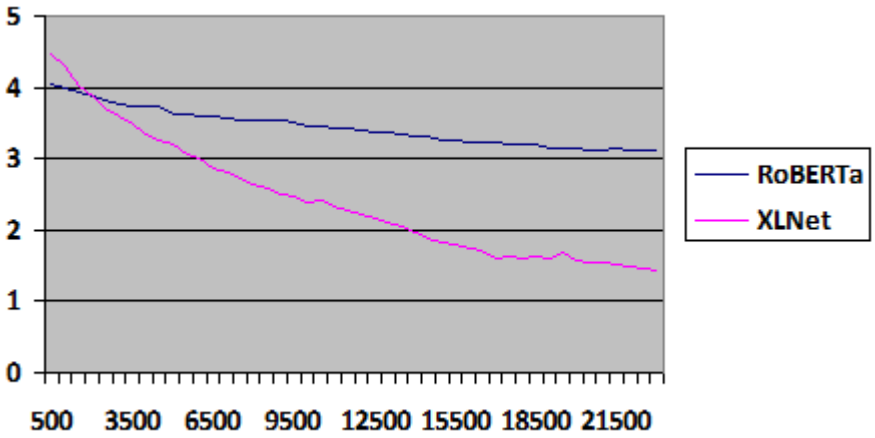

Figure 4.11: Training Loss for RoBERTa (Blue) and XLNet (Pink). We can observe that both models present underfitting.

This underfitting problem can be solved by **increasing the model complexity**. Both models may be *underfitting* simply because they are not complex enough to learn all the data. Both models employed were composed of 6 hidden layers and 12 heads, with a total number of parameters of 84095008. These amount of parameters is actually related to a small model.

4.0.2.6 Output from the Training Process

Finally, let us present the output file of the models. Each folder (test_RoBERTa and test_XLNet) contains the necessary files to recreate the process. Each folder should contain the vocabulary file and the merge file with the vocabulary. The remaining folders contain the runs and the stop points from previous processings.

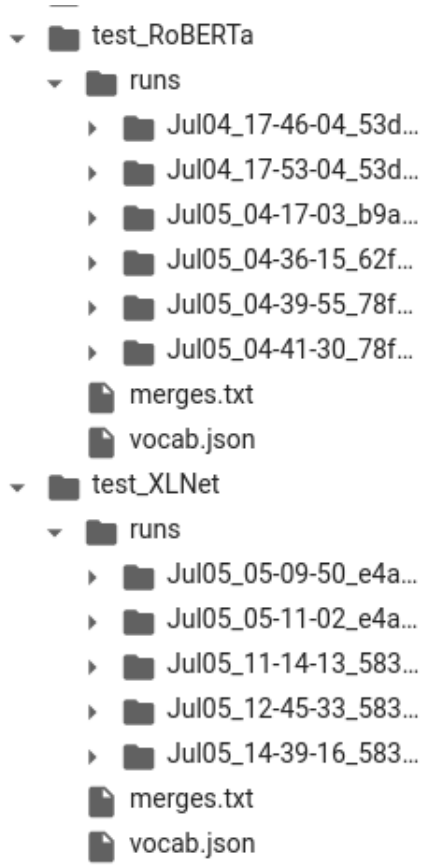

Figure 4.12: Folder distribution for training output process.

Step	RoBERTa:Training Loss	XLNet:Training Loss
500	4.035	4.456
1000	3.979	4.298
1500	3.932	4.011
2000	3.881	3.878
2500	3.823	3.701
3000	3.770	3.599
3500	3.742	3.476
4000	3.738	3.353
4500	3.727	3.266
5000	3.627	3.190
5500	3.628	3.075
6000	3.595	2.998
6500	3.601	2.879
7000	3.580	2.801
7500	3.552	2.722
8000	3.553	2.651
8500	3.528	2.589
9000	3.534	2.513
9500	3.516	2.461
10000	3.464	2.397
10500	3.453	2.403
11000	3.431	2.321
11500	3.416	2.265
12000	3.404	2.213
12500	3.370	2.154
13000	3.365	2.099
13500	3.352	2.037
14000	3.311	1.966
14500	3.303	1.884
15000	3.270	1.837
15500	3.252	1.792
16000	3.231	1.737
16500	3.221	1.712
17000	3.231	1.680
17500	3.207	1.634
18000	3.195	1.601
18500	3.210	1.633
19000	3.153	1.597
19500	3.158	1.594
20000	3.136	1.563
20500	3.132	1.536
21000	3.123	1.539
21500	3.147	1.503
22000	3.122	1.477
22500	3.126	1.461
23000	3.121	1.445

Table 4.1: Training Loss and Step for both models.

Chapter 5

Conclusions

This Masters's Thesis has created a new large-medical corpus. The corpus has the proposed size (over 1 million lines and a total of 90.990 documents). These documents are present in two different formats:

- As different plain text files separated in entity and resource folder and,
- as a single plain text file with all of the previous mentioned filed loaded and distincted with a new line character between them.

Afterwards, an analysis of two different models was done and the decision of employing RoBERTa over BERT and also using XLNet was taken. The schema followed for training these models was also created and we finally obtained two complete and but underfitted pre-trained models with a medical topic related corpus.

Finally, all developed code has been commented and documented for future use by researchers or students who want to continue this line of research.

5.0.1 Limitation of the current work

Main limitations of this work are related with time and SARS-CoV-2 pandemic lockdown. Talking and establishment process of objectives and main process of this Master Thesis followed an asynchronous communication, interfering with a smooth transmission of ideas.

Another limitation was the hardware limitations. Processes run over the computer were not feasible in the employed GPU since it always ran of memory. If the training process parameters were reduced so that this error does not rise, training process can go up to 2 months of computing. For this, the student had to use external high RAM GPU's available at Google Collab.

Finally, last limitation is related to the downloading of the corpus process. Since a lot of webpages do "not like" to be scraped, we had to insert a waiting time between different scrapings. Due to this, the time employed for downloading all of the corpus took over 2 weeks to download.

5.0.2 Future work

This Master Thesis has planted the necessary basis for the generation of different bio-related machine learning models. Although it opens a variety of work lines that will be written below, a lot of work and processing is still needed.

We will distinguish the proposed future work regarding the *Webscraping process* and regarding the model training process. Let us start with the first one mentioned:

- *More and optimized filtering methods for URLs*: Actual filtering process is done manually. The majority of webs were analysed one by one in order to extract where the highest amount of relevant information was present. This is quite a tedious and slow process, which is not optimal. A lot of URLs have a certain structure, and the author of that online article may follow a structure (information in the same tags with the same attribute names), but some pages with large databases may be built by different people who do not follow the same format, leading to scraping errors. Also, 14 different filters were made and retrieve fairly good results. But the ideal would be a scraping filter that matches perfectly each *webpage*, which is a almost impossible task due to the large amount of sources.
- *Inspecting process for the padding webs*: Webpages present in the original pool of URLs are read beforehand and checked by a professional. Those webpages are certain to contain relevant information, but padded webpages from the process are not verified. Although the padding process takes advantage of *google engine crawling process* and it assures that the retrieved webpage is one of the most useful, we are trusting an external process without internal error resolution system. For example, we may search for *Penicilin* and the retrieved webpage may be selling penicilin products, which contains the entity but not what we were searching for. FOr this, it may be beneficial to create a method that checks if the padded page will contain relevant information or not.
- and more...

After presenting these 2 different possible future work, let us enumerate and describe briefly another three proposed points for future work, but related to the model training process.

- *Test and evaluate properly the model*: Models created (RoBERTa and XLNet) were not properly evaluated, just trained. So accuracy values can't be extracted. It is preferable that the models are tested so that we know they were correctly created.
- *Employ embeddings for semantic distance between medical terms*: This is the main interest of COpenMed project. This thesis initial aim was centered in obtaining the embeddings and employing them but due to the complexity of the corpus creation and the need of perfecting the baseline of all the future work, it was preferred to reduce the goal.
- *Different tasks*: Corpus created was simply plain text. We have seen before a multitasking benchmark called GLUE. It is interesting to use GLUE as a base for creating a similar benchmark for biomedical-related tasks, such as question answer task.

Bibliography

[1] Aguilar, P. L., Glozman, M., Grondona, A., & Haidar, V. (2014). ¿ Qué es un corpus?. Revista de la Carrera de Sociología, 4(4), 35-64.

[2] del Arco, F. M. P., Valdivia, M. T. M., Zafra, S. M. J., González, M. D. M., & Cámara, E. M. (2016). COPOS: corpus of patient opinions in Spanish. Application of sentiment analysis techniques. Procesamiento del Lenguaje Natural, 57, 83-90.

[3] Asghar, N. (2016). Yelp dataset challenge: Review rating prediction. arXiv preprint arXiv:1605.05362.

[4] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., & Ives, Z. (2007). Dbpedia: A nucleus for a web of open data. In The semantic web (pp. 722-735). Springer, Berlin, Heidelberg.

[5] Beltagy, I., Lo, K., & Cohan, A. (2019). Scibert: A pretrained language model for scientific text. arXiv preprint arXiv:1903.10676.

[6] Canete, J., Chaperon, G., Fuentes, R., & Pérez, J. (2020). Spanish pre-trained bert model and evaluation data. Pml4dc at iclr, 2020.

[7] Chen, F. P., Chang, C. M., Hwang, S. J., Chen, Y. C., & Chen, F. J. (2014). Chinese herbal prescriptions for osteoarthritis in Taiwan: analysis of national health insurance dataset. BMC Complementary and Alternative Medicine, 14(1), 1-8.

[8] Cherednichenko, O., Kanishcheva, O., Yakovleva, O., & Arkatov, D. (2020). Collection and Processing of a Medical Corpus in Ukrainian. In COLINS (pp. 272-282).

[9] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. Journal of machine learning research, 12(ARTICLE), 2493-2537.

[10] Dai, J., He, K., & Sun, J. (2015). Boxsup: Exploiting bounding boxes to supervise convolutional networks for semantic segmentation. In Proceedings of the IEEE international conference on computer vision (pp. 1635-1643).

[11] Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., & Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. arXiv preprint arXiv:1901.02860.

[12] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

[13] Dolan, W. B., & Brockett, C. (2005). Automatically constructing a corpus of sentential paraphrases. In Proceedings of the Third International Workshop on Paraphrasing (IWP2005).

[14] Inoue, D., Yoshida, K., Yoneda, N., Ozaki, K., Matsubara, T., Nagai, K., ... & Zen, Y. (2015). IgG4-related disease: dataset of 235 consecutive patients. Medicine, 94(15). Kadi, H., Rebbah, M., Meftah, B., & Lézoray, O. (2021). Medical decision-making based on the exploration of a personalized medicine dataset. Informatics in Medicine Unlocked, 23, 100561.

[15] Kadi, H., Rebbah, M., Meftah, B., & Lézoray, O. (2021). Medical decision-making based on the exploration of a personalized medicine dataset. Informatics in Medicine Unlocked, 23, 100561.

[16] Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., ... & Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. Proceedings of the national academy of sciences, 114(13), 3521-3526.

[17] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

[18] Koopman, M., Zanen, P., Kruitwagen, C. L., van der Ent, C. K., & Arets, H. G. (2011). Reference values for paediatric pulmonary function testing: The Utrecht dataset. Respiratory medicine, 105(1), 15-23.

[19] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25, 1097-1105.

[20] Lai, G., Xie, Q., Liu, H., Yang, Y., & Hovy, E. (2017). Race: Large-scale reading comprehension dataset from examinations. arXiv preprint arXiv:1704.04683.

[21] Lample, G., & Conneau, A. (2019). Cross-lingual language model pretraining. arXiv preprint arXiv:1901.07291.

[22] Lee, J., Yoon, W., Kim, S., Kim, D., Kim, S., So, C. H., & Kang, J. (2020). BioBERT: a pre-trained biomedical language representation model for biomedical text mining. Bioinformatics, 36(4), 1234-1240.

[23] Luo, Y. F., Sun, W., & Rumshisky, A. (2019). MCN: a comprehensive corpus for medical concept normalization. Journal of biomedical informatics, 92, 103132.

[24] Maas, A., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011, June). Learning word vectors for sentiment analysis. In Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies (pp. 142-150).

[25] Marimon, M., Vivaldi, J., & Bel Rafecas, N. (2017). Annotation of negation in the IULA Spanish Clinical Record Corpus. Blanco E, Morante R, Saurí R, editors. SemBEaR 2017. Computational Semantics Beyond Events and Roles; 2017 Apr 4; Valencia, Spain. Stroudsburg (PA): ACL; 2017. p. 43-52.

[26] Marimon, M., Gonzalez-Agirre, A., Intxaurrondo, A., Rodriguez, H., Martin, J. L., Villegas, M., & Krallinger, M. (2019, September). Automatic De-identification of Medical Texts in Spanish: the MEDDOCAN Track, Corpus, Guidelines, Methods and Evaluation of Results. In IberLEF@ SEPLN (pp. 618-638).

[27] Martin, L., Muller, B., Suárez, P. J. O., Dupont, Y., Romary, L., de La Clergerie, É. V., ... & Sagot, B. (2019). Camembert: a tasty french language model. arXiv preprint arXiv:1911.03894.

[28] McCann, B., Keskar, N. S., Xiong, C., & Socher, R. (2018). The natural language decathlon: Multitask learning as question answering. arXiv preprint arXiv:1806.08730.

[29] Névéol, A., Grouin, C., Leixa, J., Rosset, S., & Zweigenbaum, P. (2014). The QUAERO French medical corpus: A ressource for medical entity recognition and normalization. In In proc biotextm, reykjavik.

[30] Pampari, A., Raghavan, P., Liang, J., & Peng, J. (2018). emrqa: A large corpus for question answering on electronic medical records. arXiv preprint arXiv:1809.00732.

[31] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. arXiv preprint arXiv:1802.05365.

[32] Pires, T., Schlinger, E., Garrette, D. (2019). How multilingual is multilingual BERT?. arXiv preprint arXiv:1906.01502.

[33] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training.

[34] Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., ... & Sutskever, I. (2021). Learning transferable visual models from natural language supervision. arXiv preprint arXiv:2103.00020.

[35] Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text. arXiv preprint arXiv:1606.05250.

[36] Recht, B., Roelofs, R., Schmidt, L., & Shankar, V. (2018). Do cifar-10 classifiers generalize to cifar-10?. arXiv preprint arXiv:1806.00451.

[37] Rönnqvist, S., Kanerva, J., Salakoski, T., & Ginter, F. (2019). Is multilingual BERT fluent in language generation?. arXiv preprint arXiv:1910.03806.

[38] Sang, E. F., & De Meulder, F. (2003). Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. arXiv preprint cs/0306050.

[39] Sass, J., Bartschke, A., Lehne, M., Essenwanger, A., Rinaldi, E., Rudolph, S., ... & Thun, S. (2020). The German Corona Consensus Dataset (GECCO): a standardized dataset for COVID-19 research in university medicine and beyond. BMC Medical Informatics and Decision Making, 20(1), 1-7.

[40] Tian, Y., Ma, W., Xia, F., & Song, Y. (2019, August). ChiMed: A Chinese medical corpus for question answering. In Proceedings of the 18th BioNLP Workshop and Shared Task (pp. 250-260).

[41] Ushizima, D., Carneiro, A., Souza, M., & Medeiros, F. (2015, December). Investigating pill recognition methods for a new national library of medicine image dataset. In International Symposium on Visual Computing (pp. 410-419). Springer, Cham

[42] Wang, Y., & Bai, Y. (2007). A corpus-based syntactic study of medical research article titles. System, 35(3), 388-399.

[43] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2018). GLUE: A multi-task benchmark and analysis platform for natural language understanding. arXiv preprint arXiv:1804.07461.

[44] Grabar, N., & Cardon, R. (2018, November). Clear-simple corpus for medical french. In ATA.

[45] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2018). GLUE: A multi-task benchmark and analysis platform for natural language understanding. arXiv preprint arXiv:1804.07461.

[46] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., & Le, Q. V. (2019). Xlnet: Generalized autoregressive pretraining for language understanding. Advances in neural information processing systems, 32.

[47] Zbontar, J., Knoll, F., Sriram, A., Murrell, T., Huang, Z., Muckley, M. J., ... & Lui, Y. W. (2018). fastMRI: An open dataset and benchmarks for accelerated MRI. arXiv preprint arXiv:1811.08839.