BIOMEDICAL ENGINEERING MSc PROGRAM
UNIVERSITY OF CRETE
TECHNICAL UNIVERSITY OF CRETE
FOUNDATION FOR RESEARCH AND TECHNOLOGY-HELLAS

Application Grade Thesis

A Machine Learning method to classify sentences containing biomedical entities in academic text

Student's Name: Apostolos Boumpakis

Supervisor's Name: Alexandros Kanterakis

Date of completion: 24/02/2022

This dissertation is submitted as a partial fulfilment of the requirements for the Master's degree of Biomedical Engineering M.Sc. program

Abstract

Nowadays, the amount of biomedical literature is getting larger and larger and thus Natural Language Processing (NLP) research in clinical documents is gaining a very significant role. The automated analysis of biomedical literature is rapidly growing, stimulating the development of several techniques of automatic Named Entity Recognition (NER) and document classification. However, despite the existence of so many techniques for the classification of biomedical entity sentences, few types of entities can be easily recognized.

The aim of this study is to recognize Disease, Gene, SNP and Chemical entities from biomedical texts, using Bidirectional Encoder Representations from Transformers (BERT), a new and advanced Named Entity Recognition technique.

Briefly, hundreds of biomedical papers were parsed and analyzed to their sentences, classified and labeled accordingly, in order to create different datasets. Finally, they were passed through the BERT model in order to sort the sentences that include the aforementioned entities.

The results showed that by appropriately pre-training the BERT model, great recognition performance can be achieved, without extensive fine-tuning and optimization requirements, while outperforming previous models on NER biomedical text mining task. However, there is by all means space for further tuning and much more future work and new challenges.

Table of contents

List of figures

List of tables

Chapter 1: Introduction

To date, a large amount of biomedical content has been published for clinical documents. Millions of articles are published each year about new discoveries in the medical field. That is why it is of considerable significance to conduct Natural Language Processing (NLP) [1] research in biomedical literature. The automated analysis of biomedical literature has rapidly grown in research during the last years. This area of research has stimulated the development of several techniques of automatic Named Entity Recognition (NER) and document classification. Nevertheless, many open issues must be addressed in order to obtain satisfactory results. Despite the existence of a plethora of methods for classification of sentences containing biomedical entities and entities with deep Neural Network frameworks, it is an open question which types of entities can be easily recognized.

The final goal of this study is to present the state-of-the-art Named Entity Recognition technique, Bidirectional Encoder Representations from Transformers (BERT) [2], in order to recognize/extract *Disease*, *Gene*, *SNP* and *Chemical* entities from biomedical texts. The reason why BERT was chosen is the fact that it is the most widespread Neural Network architecture for training language models, having led to considerable improvements in various NLP tasks.

In general, the more the parameters in a BERT model, the better the results obtained. Unfortunately, due to the fact that the memory consumption increases with the size of these models, the lighter BERT variant, distilBERT [3,4], was applied. This technique was evaluated on two NER tasks for each entity.

All in all, in outline, hundreds of biomedical papers were parsed in an XML format, analyzed to their sentences, classified and labeled accordingly, so as to create different datasets. Finally, they were passed through the BERT model to recognize sentences that include (or not) specific entities.

The study proceeds as follows: Chapter 2 analyzes the background and the literature about the investigated topic, underlying the original contribution of this research and making the reader familiar with the existing key theories. Chapter 3 describes the methodology carried out for the study and the justification behind this research design. Chapter 4 is the results chapter, where the data collection results are presented. Chapter 5 interprets the data and discusses the results obtained. Chapter 6 states the challenges and suggests potential future work. Finally, Chapter 7 concludes the study. It ties it together and highlights the key takeaways.

Chapter 2: State-of-the-art

2.1 NLP

Nowadays, the reality of having machines talk and respond like humans keeps getting more and more plausible. People you ask for queries on websites, smart assistants (such as Cortana, Siri) etc. are not human. But how do they manage to seem so human-like, so much intelligent? This is where Natural Language Processing (NLP) comes in, which refers to the Artificial Intelligence method of communicating with an intelligent system using the natural language [1]. In other words, it gives the machines the ability to understand text and spoken words.

Enormous quantities of data are transferred due to the everyday use of social media by humans, providing useful information about human behavior and habits, so that machines can mimic human linguistic behavior. Some of the applications of NLP are the following [5]: First of all, autocorrect, which automatically corrects a misspelled word. Then, we have customer chat services, for example chatbots or assistants like Cortana, as previously mentioned. Machine translation is also another use-case of NLP, with Google translate being the most common example for it.

Various steps are involved in order to transform a text into a form comprehensible by the machine [6]. Firstly, *segmentation* needs to be performed. *Segmentation* is the process of breaking the whole text into its constituent sentences. After that, each sentence is divided into tokens, constituent words. The next process in NLP is *stemming*, which is the process of reducing the words to their root form by cutting off the beginning and the end of the word, taking into account a list of common prefixes and suffixes. Another way to identify the base words is *lemmatization*, where words are mapped to their actual form via a dictionary used by transformation. Once the tokens are divided into their root form, their part of speech (POS) is determined as many words, especially common ones, that can serve as multiple parts of speech, subject to the situation in which they are used. Next, the machine is introduced to *Named Entity Recognition (NER)*, which is the process of determining which items in the text map to proper names, like places or people, and what the class of each such name is (e.g., person, location, organization).

Once base words and tags are defined, a Machine Learning algorithm is used in order to teach the model human sentiment and speech. In order to execute all these programs and all of this function on a given text file, Python programing language has come up with the Natural Language Toolkit (NLTK), an open-source collection of libraries, programs, and education resources for building NLP programs [1]. The NLTK includes libraries for many NLP tasks, plus libraries for subtasks, such as *sentence parsing*, *word segmentation*, *stemming* and *lemmatization* and *tokenization*. It also includes libraries

for implementing capabilities, such as semantic reasoning, the ability to deduce rational assumptions centered on facts extracted from text. It has over 100 corpora and a lot of pre-trained models, as well. Other popular Python libraries are Textblob, which is used for processing text-based data and which provides simple APIs, Spacy, which is another common library used for advanced NLP and works well with deep learning frameworks, Gensim, Corenlp and many others. All the above libraries provide all the aforementioned features as well (*tokenization*, *stemming*, etc.).

2.1.1 Biomedical NLP

As previously mentioned, with the recent statistical and neural revolutions, the research community has made great strides in many fronts, such as question answering, machine translation, speech recognition, etc. However, the mainstream NLP focuses on general domains, such as the web. In contrast, specialized domains, such as biomedicine have received relatively little attention.

As a field of research, biomedical NLP incorporates ideas from NLP, bioinformatics, medical informatics and computational linguistics [7]. Thanks to the rapid digitization of health records, the growing number of biomedical publications, and the escalating presence of health information online, biomedicine is taking more and more steps towards precision medicine, where treatments become increasingly effective by tailoring to individual patients. Automatic extraction of molecular mechanisms NLP can play a key role in this revolution, as a large quantity of biomedicine is conducted and recorded in natural language [8]. For example, PubMed alone adds millions of biomedical papers every year. PubMed is a collection of biomedical research literature available to read on the web [9]. Biomedical text is drastically different from general-domain one.

Although annotated corpora, used in the development and training of general purpose, may provide evidence of general text properties such as parts of speech, there is a word distribution shift from general domain corpora to biomedical corpora. Biomedical named-entity recognition (BM-NER) plays an essential role in biomedical language processing: terms of interest are identified and mapped to a pre-defined set of semantic categories [10]. Examples of BM-NER systems include identifying diseases and drug names, and detecting *Gene*, *Protein*, or *Disease* mentions in biomedical papers. BM-NER is getting better with the rise of more annotated corpora to learn from. Publicly available tools specific for biomedical literature research include PubMed search, Europe PubMed Central search, GeneView and APSE. The unstructured components of the Electronic Medical Records (EMRs) and the Electronic Health Records (EHRs) collected during a diagnosis are often free-text and difficult to search. Thus, in order for them to be analyzed, various tools have been developed, such as the MedLEE system (for chest radiology reports analysis), or PubTator, cTAKES, and CLAMP (for clinical text annotation).

2.1.2 Pubtator

Manually extracting useful information (e.g., annotations) from biomedical literature and turning it into structured databases is a very expensive and time-consuming task. That is why, there is the need of automated text mining tools. PubTator is a free access web-based system, which provides automatic annotations of biomedical concepts, such as *Genes*, *Diseases*, *Species*, *Chemicals* and *Mutations* [11]. It includes about 3 million of the full-text articles in the PMC Open Access subset and about 30 million of abstracts in PubMed. It is a tool perfectly suited for biocurators (i.e., researchers who collect and annotate information which is distributed by biological databases) with little text-mining experience, due to its various unique features compared to other tools.

First of all, being a web-based system, it is available to all computer platforms, without the need for installation or maintenance. Moreover, a user who is familiar with PubMed will have no problem using PubTator, since its interface is very close to PubMed. Thirdly, it can very efficiently generate automatic computer pre-annotations in computer-assisted extraction of knowledge from unstructured biological data into a computable form (biocuration). Finally, with PubTator, the user is able to either search and retrieve articles or input a list of PubMed articles and by the completion of the biocuration, they can download and export the annotations for database integration in multiple formats (XML, JSON and tab delimited).

Some entity recognition tools that ensure the automatic process of results are GeneTUKit for *Gene* mention, GenNorm for *Gene* normalization, SR4GN for *Species*, DNorm for *Diseases*, tmVar for *Mutations* and a dictionary-based lookup approach for *Chemicals* [12].

A significant drawback of PubTator is the fact that it only provides automated concept annotations across abstracts. However, since 2019, PubTator Central (PTC) has made its appearance, which is able to expand the automated concept annotation to full text articles. It includes a web interface designed for full text, updated concept annotation methods, and an expanded set of concept types. Annotations in PTC are also available in XML, JSON and tab delimited format via the online interface, a RESTful web service (API) for programmatic access or FTP download in bulk. PTC adds new articles every day to always keep in synchronization with PubMed and PMC.

2.1.3 BioC

Although NLP tools are becoming more and more sophisticated, there still exists the challenge of dealing with more complex systems. Also, according to their expertise, researchers write specific software in order to process data in various formats. However, this results in hardly adaptable systems and of limited use. A solution towards the aforementioned issues has to do with the discovery of BioC,

which is a simple format to represent, store and share text data, offering broad use and reuse. It is a straightforward data structure for dealing with text and annotations, in order to achieve interoperability and it can easily handle documents of various lengths, abstracts or full-text articles and different annotations, including tokens, sentences, named entities (*Genes*, *Chemicals* etc.) [13]. The main goal of BioC is to provide an ease in sharing the results of an algorithm, so as to get rid of the most essential impediment, which is the reuse of the tools, and thus support the development of text mining systems designed for different workflows. Developers use Windows, Linux, Mac and work in C, C++, C#, Python and so forth. That is why interoperability plays, too, a crucial role and requires an unhindered connection among data.

In order to deal with the problems mentioned above and especially interoperability, an XML type definition (DTD) is used [9]. XML elements contain "infon" elements that store key-value pairs with any desired semantic information. Semantics are defined via a "key" file, which is a simple text file, where the developer defines the semantics associated with the data. Key files may describe data not seen before, that is why BioC users need to manually add new key files, in order to represent their BioC data collections.

Another model that makes it possible to analyze biomedical text and annotations is the Resource Description Framework (RDF), a standard model for data interchange on the Web. Nonetheless, XML is better, and more widely used. Be that as it may, a combination between those two could possibly lead to even better results.

There are dedicated libraries for BioC XML that populate and preserve native BioC classes, or data structures, in a number of different programming languages, thus there is no requirement for internal or external XML format knowledge. This means that text and annotations can easily be shared between various tools. PubMed Central articles are available as large File Transfer Protocol (FTP) files in order to be able to obtain a large number of articles. Apart from that, it is possible to download exactly the articles needed, thanks to a RESTful web service, making it easier for updating a large collection.

The BioC process sequence is the following: First, the data is read into BioC data classes via an Input Connector and the XML input is either a file or from a web server. Second, the Data Processing module, which is any kind of NLP process. Third, the Output Processor, where the output data from the BioC class is passed, in order to produce new data in the BioC format.

The basic elements of a BioC XML file are the following:

- *Collection of documents*: The BioC XML file format starts with a collection of documents. It is a set of documents with information about the original source and the creation date. The documents consist of a succession of passages that include all the sections of a journal article.
- *Sentence segmentation*: Each passage consists of a set of sentences instead of the text of the passage, as their distinction is important for NLP tools.
- *Text annotations*: Text annotations refer to the input and output for biomedical text processing programs. Some biomedical examples include *Genes*, *SNPs* and *Chemicals*. The location tag connects this information with the original text. An annotation is usually one continuous segment of text. However, multi-segment annotations can also be represented. Moreover, the attribute "id" can be added, allowing the annotation to be referenced in relations.
- *Relation annotations*: Relation annotations refer to the relation between entities and other features (e.g., *Gene–Chemical* correlations) by specifying a set of annotations which participate in the relation and the role of each item in this. Once again, by adding the attribute "id", a relation can be a member of other relations.

As already mentioned, an important feature of the BioC XML format is the "infon", which keeps a key-value pair with any desired semantic information. Key files define the "key" strings and the "value" strings. A passage infon with key="type" signals values such as "Introduction" or "Methods" of the whole text. An annotation infon with key="type" could indicate "*Gene*", "*Chemical*" etc.

2.2 Deep Neural Network in Biomedical NLP

2.2.1 Machine Learning

The main difference between humans and computers is that humans learn from past experiences, whereas computers need to be told what to do, to follow instructions. In order to get computers to learn from experience too, Machine Learning was discovered. Of course, past experiences for humans correspond to past data for computers. Machine Learning is an application of Artificial Intelligence that provides systems the power to learn and improve from their own experience, without being explicitly programmed [14]. Diving in a little deeper, we can see that the process of learning begins with the analysis of the input data, such as examples or instructions and then the system tries to find patterns like shapes, size, color etc. Based on these patterns, the system tries to predict the different types of the input data and segregate them. Finally, it keeps track of all such decisions it took in the process, in order to make sure it is learning. Thus, the next time the system is asked to predict and segregate the different types of input data, it will not have to go through the entire process again.

The three primary types of Machine Learning [15] are, first of all, supervised Machine Learning, which

involves supervising a model, as it trains to fit on a set of trained data. Labeled training data is required in order to predict the future, by applying what has been learned in the past. The learning algorithm can also compare the extracted output with the correct one and find errors, so as to modify the model accordingly. This kind of a model is generally used into filtering spam mails from email accounts for example. In contrast, unsupervised Machine Learning algorithms consist of training data, but unlabeled ones, so as for systems to infer a function for describing a hidden structure. Finally, there is reinforcement learning, where the system learns on its own by interacting with its environment and discovering errors and rewards. In other words, it is mostly based on trial and error and delayed reward. This method could lead to maximum performance, since machines are trained to automatically determine the ideal behavior within a specific context.

One should be very careful when selecting the kind of solution for a model, in order not to lose time and processing cost. The factors that help to the selection of the right kind of Machine Learning solution are the following: First, the statement of the problem, meaning the description of the model that will be built. A very good perception of the problem can lead to the choice of the right algorithm. Then comes the size, quality and nature of the data. For example, large and categorical data lead to supervised learning solutions. Finally, another factor that needs to be taken into account is the complexity of the algorithm. If a problem can be dealt with supervised learning solutions, there is no need to apply reinforcement learning, as this would be very difficult and time consuming for no reason.

Three common types of Machine Learning problems are *classification*, *regression* and *clustering* [16]. A *classification* problem involves predicting whether a given observation belongs to a certain category. Based on earlier observations of how the input maps to the output, *classification* "guesses" a classifier which can generate an output for arbitrary input. A classifier can then label an unseen example with a class. For example, after a set of clinical examinations that relate vital signals to a disease, one can predict whether a new patient with an unseen set of vital signals suffers that disease and needs further treatment. The algorithms that fall on the *classification* are Naive Bayes, KNN, Logistic Regression and Random Forest. Moving on, a *regression* problem is a kind of Machine Learning problem that tries to predict a continuous value for an input based on previous information. The input variables are called the predictors and output the response. *Regression* is similar to *classification*, as it also estimates a function that maps input to output, based on early observations. But this time an actual value is estimated, not just a class of an observation. There are many different applications of *regression*, such as the estimation of the chances of getting hired for a specific job based on the university grades. Finally, in *clustering*, the goal is to group objects that are similar in clusters, while making sure that the

clusters themselves are not similar. *Clustering* is different in the sense that you don't need any knowledge about the labels. Also, there is no right or wrong. Different *clustering* reveals different information about the objects. A popular *clustering* method is k-Means, which clusters the data in k clusters, based on some similarity measure.

There is an alternative and shortened definition of Machine Learning: "The use of data in order to answer questions" [17]. Using data is what we refer to as training and answering questions is what we refer to as making predictions. Training refers to using the data, in order to fine-tune a predictive model, which is then used to make predictions on previously unseen data. The more the data, the better the model.

Let's now put together the 5 main areas of the Machine Learning process [18]:

- Data collection and preparation: This step includes the whole process from choosing where to get the data, up to the point it is ready for feature engineering. During data preparation, we load the data in order to use it in Machine Learning training. Moreover, data needs to be split into two parts. The first part is used for the training of the model and is the majority of the dataset. The second part is used for the evaluation of the trained model's performance. This is a very important step, since the quality and quantity of the data determine the performance of the model.
- *Feature selection and feature engineering*: This step includes all the changes applied to the data, from the clean up until its importation into the Machine Learning model.
- *Selection of the Machine Learning algorithm and training of the first model*: Training is considered the bulk of Machine Learning. In this step, data is used to incrementally improve the model's performance. In a small scale, the formula for a straight line is $y=m \cdot x+b$. The variable x is the input, y is the value of the line at position x, m is the slope of the line and b is the y-intercept. The m and b values are used for training, affecting the position of the line. In Machine Learning, there are many features, meaning that there are many m_s as well, which all together form a matrix, denoted as W and called weight matrix. The same applies for the b values, which are called biases.

Before training, some random values are initially assumed for W and b, in order to predict the output during the training process. As it is expected, poor results are extracted. After that, the model's predictions are compared to the expected output and the values in W and b are adjusted, so as to produce more correct predictions. This process is then repeated and after each iteration, the weights and the biases are accordingly adjusted.

- *Model evaluation*: After training, the model is tested against data that has never been used for training, so as to evaluate how well the model performs against data not seen before. Usually, a

training-evaluation split is somewhere in the order of 80/20 or 70/30. depending on the size of the original source dataset. A lot of data means no need for a big fraction for the evaluation dataset.

- *Model tweaking, regularization, and hyperparameter tuning*: This is where the model is iteratively further improved by tuning the training parameters. One example, leading to higher accuracies, is by running through the training dataset multiple times. Another tuning parameter is the learning rate, which determines the step size at each iteration while moving towards a minimum of a loss function, based on the information from the previous training step.

It is important that a good model is defined from the start, otherwise it might take too much time to make small changes in order to improve the model.

These parameters are referred to as “hyperparameters”. They are parameters that work at a higher level and control the behavior of the learning algorithm. The tuning of these hyperparameters can be set by the designer or they can also be learnt through the process of validation. It is an experimental process that depends on the specifics of the dataset, model, and training process.

2.2.2 Deep Neural Network

Artificial Neural Networks (ANN) are finite specific part of Machine Learning. An ANN is a function that transforms input data to output data based on training over many correct examples [19]. It is a network that is a collection of neurons connected together. Each neuron acts as a processing unit, collects input, computes its weighted sum and activates the output. Although individual neurons are simple, the entire neural network exhibits great processing potential. If this is done correctly, then flexible, nonlinear systems will be created, which might exhibit “artificial intelligence”, such as the one of the human brain. Early networks used a function called sigmoid function, in order to squish the relevant weighted sum into the interval between zero and one, motivated by the biological analogy, either being inactive or active. However, few modern networks use sigmoid anymore. Nowadays, ReLU is very popular, as it seems much easier to train [20]. ReLU stands for Rectified Linear Unit and has an activation function of the form depicted in Fig. 1. So, below zero, the output is zero and above zero the output is linear. Thus, it is a piecewise linear function with a very simple implementation. Moreover, activation functions that lead to probability distributions can be defined. These are known as SoftMax Neurons. The key idea is that if I have a collection of numbers in my output, where the higher the number the higher the probability for the output to correspond to this particular output, using the SoftMax equation:

$$y_l(\mathrm{k}) = \frac{e^{z_l(k)}}{\sum_{i=1}^{d_l} e^{z_l(i)}}$$

These numbers can be transformed into a probability distribution. The output values are between zero and one and sum to one and the output of a neuron depends on the outputs of all the other neurons in its layer.

Neural Networks are not isolated neurons. We have to think of a global structure. Typically, in order to form a NN, neurons have to be organized in layers. In its simplest form, we have a layer of neurons, the output layer and an input layer which is a layer of input values. These values are fed into the neurons and each connection has its own weight. The real layers are the weights, also called as hidden layers and not the inputs and the outputs. Three or more layers are able to create complex shapes.

Two very famous network topologies are the Feedforward NN (FNN) and the Recurrent NN (RNN) [20,21]. The FNN starts with the input layer. Information is propagated through all hidden layers and eventually arrives to the output layer. There are no connections between neurons in the same layer and from one layer to the previous one. This does not apply for the RNN. In this case, the processing capability is much higher, but on the other hand, the entire system may exhibit complicated dynamics and be difficult to train. In practice, its architecture is like having an FNN that unrolls over time. A single neuron receives an input, produces an output and sends its own output to another neuron that receives the next input and combines it to compute the output and also sends its output to the next neuron and so on (see Fig. 2). Thus, having just one neuron with the recurrent connection is more or less equivalent to having multiple neurons implementing some kind of recurrent relation.

As far as the learning process of NNs is concerned, its structure is predefined, meaning that neurons cannot be added or removed as the network is trained. So, typically, we start with a fixed number of neurons and layers and then we try to adjust the parameters of the model. So, in the learning

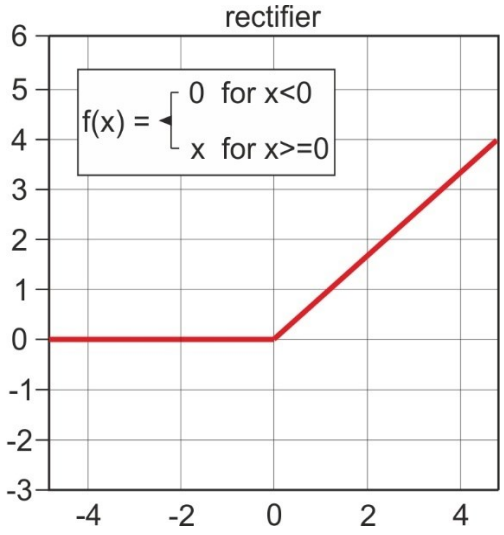


Fig. 1: Rectified Linear Unit (ReLU) activation function.

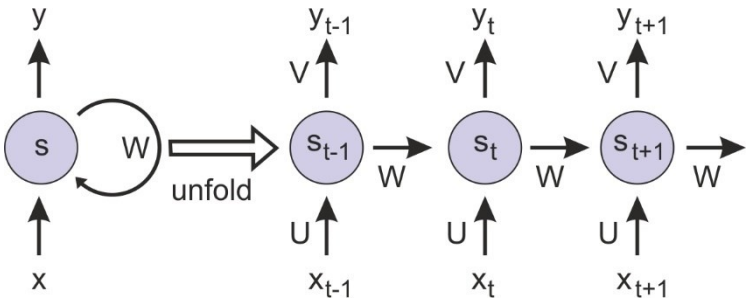


Fig. 2: Recurrent Neural Network architecture.

process, we try to fix just the parameters, but this is done using labeled training examples. That means that, for specific inputs, we know the correct output and we present those examples to the network. This process is done for several thousands of examples and since these are labeled correctly, the network is, in a sense, taught what we want to do. This is called supervised learning as already mentioned.

Now let's move on to deep NNs and see what "deep" means. The simple answer is that deep means more than two hidden and output layers. So, strictly speaking, it is not something "new". The reason why we need at least two layers is because we have a universal approximator. The universal approximator theorem states that "A feed-forward network with a single hidden layer containing a finite number of neurons, can approximate continuous functions on compact subsets of R_n, under mild assumptions on the activation function." [22]. Recent theoretical results show that there is indeed some power into using deep architecture with many hidden layers, because the same function can be represented with exponentially fewer units. The reason why DNNs came into life during just the last decade and not earlier has to do with the fact that CPUs are now much more powerful, the GPUs appeared to support graphics computations but, in a sense, they are also used for the parallel computations required for training deep architectures and finally the fact that big data also appeared.

RNNs have had a lot of success with natural language data, but they do have some drawbacks. Two of the biggest drawbacks are that each sequence needs to be "seen" in order, which means that there is a limit to how much you can parallelize the processing, breaking apart the beginning and the end of the sequence and then handling those on different machines. As a result, there is a bottleneck to how fast you can train these models. Another drawback is that it is easier for them to capture relationships between points that are close to each other than it is to capture relationships between points far from each other.

2.2.3 Transformers

In order to help address some of these problems, Transformers were proposed [23]. Transformers are models that can translate text to even generate computer code. They are also currently being used in biology in order to solve the protein folding problem.

In general, Transformers are a type of NN architecture and until they came around, the way deep learning was used to understand text was with the RNN model. Let's say we translate a sentence from English to Greek. An RNN takes as input an English sentence and processes the words one at a time and then sequentially extracts their Greek counterparts. However, as already mentioned, RNNs have problems with large sequences of text and they are hard to train. This is where the Transformers appeared, initially as their only purpose to do translation. Unlike RNNs, Transformers can easily be parallelized, meaning that huge models can be trained.

There are three main innovations that make this model work so well: Positioning, Attention and Self-Attention. To begin with, Positional Encodings is the idea that instead of looking at words sequentially, each word of the sentence is given a number before being fed into the NN, depending on its order in the sentence. In other words, information about word order is stored in the data itself rather than in the structure of the network. Then, as the network is trained on lots of text data, it learns how to interpret these positional encodings. More specifically, the positional encoder is a vector that has information on distances between words. The English sentence is passed through the input embedding [24] and the vector encoding of position in sentence is applied, thus getting word vectors which have positional information, the context. So, the NN learns the importance of word order from the data. The next innovation, the Attention mechanism, is a NN structure that allows the text model to look at every word in the original sentence when making a decision about how to translate a word in the output sentence. The model knows which words it attends to by learning over time from data. By seeing thousands of examples of English to Greek sentence pairs, the model learns about word order, gender and generally all the grammatical stuff. However, the real innovation of the last years is Self-Attention. Self-Attention is not just trying to align words, but instead it tries to understand the underlying meaning in language, so as to build a network that can do any number of language tasks. Attention can be a very effective way to get a NN to understand language if it is turned on the input text itself. Self-Attention allows a NN to understand a word in the context of the words around it. It can also help NNs distinguish the meaning of the same word between different sentences, or recognize parts of speech. For each word, we have an attention vector that captures contextual relationships between words in the sentence. Self-Attention looks at the relationship between each

item in the input sequence and every other item in the input sequence. This is done several times, so each head learns attention relationships independently. This is called multi-headed self-attention.

To sum up, the overall Transformer architecture uses a model that has an encoder and a decoder. The encoder takes in the input sequence and transforms it into embeddings (numeric representations of the input sequence). The decoder takes this embedding, in order to create the output sequence. The encoder and the decoder are both made up of many multi-headed self-attention modules stacked on top of each other.

Before the appearance of Transformers, Long Short-Term Memory (LSTM) networks were used for language translation [25]. However, they had some serious drawbacks. They were slow to train, since words were passed in and generated sequentially, so it took many time steps for the NN to learn. Also, although the bidirectional system was applied, they were learning left to right and right to left context separately and then concatenating them, thus losing the context.

One of the most popular Transformer-based models is called BERT and it is going to be explained at the next section.

2.2.4 BERT

As time goes by, progress is rapidly accelerating in Machine Learning models that process language. This progress in natural language understanding, as applied in research, represents one of the biggest steps forward in the history of it (i.e., the research).

One very big challenge in NLP is the shortage of training data, because it applies too many distinct tasks [2]. Thus, each task datasets are very specific and contain limited training examples. However, deep learning-based NLP models exploit large amounts of data. In order to tackle this problem concerning data, researchers have developed techniques for training general purpose language representation models, using unannotated text that exists in abundance on the web. The model is then fine-tuned on small-data NLP tasks, thus improving in accuracy compared to training from scratch.

Three years ago, a new technique was developed for NLP pre-training, called Bidirectional Encoder Representations from Transformers (BERT) [2,26,27]. Thanks to this technique, one can now train their own sentiment analysis task, named entity recognition, question answering and other models in less than an hour on a single Cloud TPU, or in a few hours using a single GPU.

BERT is based on existing strategies for applying pre-training language representations, such as Generative Pre-Training Transformer and ELMo [28]. However, this is the first deeply bidirectional,

unsupervised language representation, pre-trained using only a plain text corpus [2]. BERT prevails over other contextual models, which generate a representation of each word that is based on the other words in the sentence, as it represents a word using both its previous and next context, starting from the bottom of a deep neural network, so it makes it bidirectional.

In other words, BERT models "weight" the full context of a word by looking at the words that come before and after it. BERT's neural network architecture is shown in Fig. 3 [2], depicting its differences compared to previous contextual pre-training models.

The arrows indicate the information flow from one layer to the next. The green boxes indicate the final contextualized representation of each input word.

BERT is the first model successfully used to pre-train a deep neural network. It is the result we get if we stack the encoders of the Transformers model on top of each other and make the model deeper with 12 layers (Fig. 4) [29].

BERT is trained on huge text dataset. But where does it come from? When we are doing machine translation, we have got sentences from different languages thus we have labeled training data. But if we remove the decoders, how are the encoders trained? The answer is the key contribution of BERT. That is figuring out tasks to train BERT on, for which we have tones of labeled training data.

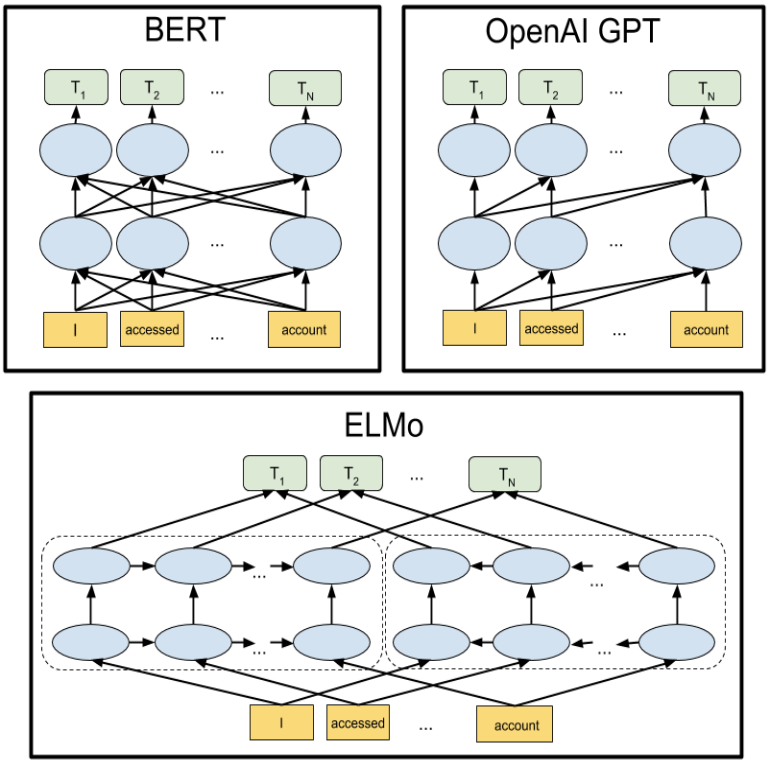


Fig. 3: BERT's neural network architecture versus previous contextual pre-training models [2].

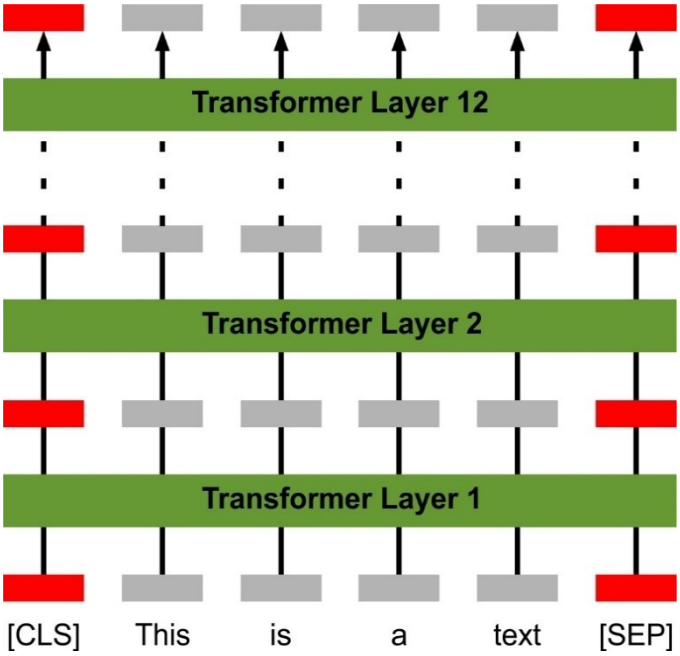


Fig. 4: BERT's neural network architecture.

So, its training consists of two steps. The first step is pre-training, so that the model can understand language and context and the second step is fine-tuning, where the model learns how to solve specific tasks, knowing the language [27].

As far as the first step is concerned, although a unidirectional model is trained conditioning each word on its previous words, this method cannot be applied to a bidirectional model. That is why the mask technique is used, where some random words of the input are masked out and then each word is conditioned bidirectionally, in order to predict the masked words/tokens.

Another task included in the pre-training model, in order to help BERT handle relationships between sentences and understand context across different sentences themselves, is next sentence prediction. BERT takes in two sentences and predicts if the second sentence follows the first.

Two sentences are given as input, with some random masked words. The embedding vector that is used as input consists of three embeddings: The token embeddings which are the pretrained embeddings, the segment embeddings, which is the sentence number encoded into a vector and the position embeddings, which is the position of a word within the sentence that is encoded into a vector. The last two embeddings account for the ordering of the inputs.

Inside an encoder, it is like having two main layers: A multi-headed self-attention layer and an FFNN [29].

Thanks to the self-attention layer, the encoder can look at all the other words around it in the input sentence as it encodes a specific word. It figures out how to incorporate those words into the encoding of this specific word. Then, the results of the self-attention are fed to the FFNN. Each word has a vertical line going through the 12 Transformer layers (see Fig. 4 above). Let's analyze what these lines represent.

We know that words are represented by embeddings. They have their own vector of length 768. Let's say that the encoder encodes one word at a time. The encoder receives a list of vectors, one vector per token and passes them into a "self-attention" layer and then into an FFNN. The output, the "enriched" embedding is sent out upwards to the next encoder and so on.

Self-attention's mechanism resembling the one RNNs are using, combines their representation of previous words (and consequently vectors) that they have processed with the current one that they are processing, by maintaining a hidden state. So, we have the creation of an attention vector, which captures contextual relationships among words in the input sentence. Eight such attention vectors are determined per word and a weighted average is taken in order to compute the final attention vector for every word.

A single-headed attention looks like this: For every single word, we have V, K and Q vectors that extract the attention vectors for it (Fig. 5). They are used in order to compute the outputs of the self-attention layer, applying the following formula:

$$Z = \text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{Dimension\ of\ vector\ Q,K\ or\ V}}\right) \cdot V$$

For multi-headed attention, we have multi-headed matrices WV, WK and WQ. So, we will have one attention vector Z_i for every word. More specifically, this self-attention calculation mentioned above is performed eight times with weight different matrices, thus ending up with eight different Z matrices.

However, the FFNN is expecting one matrix, one vector for each word. That is why these eight matrices are concatenated and multiplied by another weighted matrix, WZ, resulting in a Z matrix that is sent forward to the FFNN.

Each word's line/path is independent of the others'. Thus, we can use parallelization. That is why all the words can be passed on to the encoder block at the same time. The output is a set of encoded vectors, one for every word.

Concerning the output, [CLS] is the binary output for the next sentence prediction, producing 1 if the second sentence follows the first in context and 0 if it doesn't. It is used as a token to represent the classification of a specific input. The [SEP] tokens are used at the end of every input sequence. The Ts

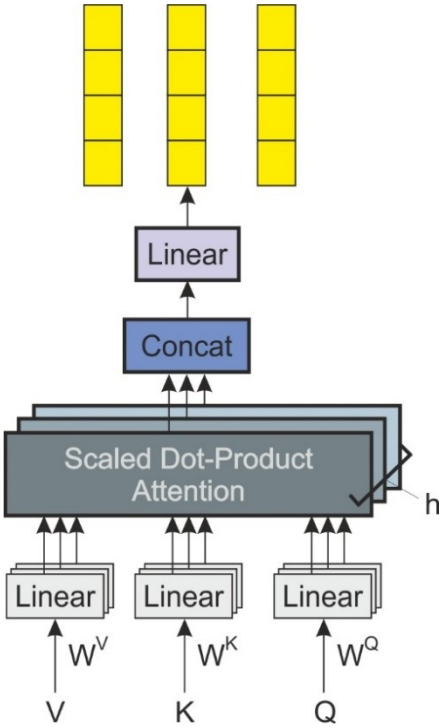


Fig. 5: Single head attention architecture.

are word vectors that are generated simultaneously and correspond to the outputs for the masked language model problem.

As far as fine-tuning is concerned, BERT can be further trained on specific NLP tasks, such as question answering. This model is trained by modifying the inputs/weights and the output layer. As inputs, the question is given along with a passage that contains the answer. The output layer outputs the start and end words that include the answer.

Since the first paper [27] was published, there has been a lot of work on BERT. The architecture has been extended to make it smaller, more compact and cheaper to run. Some popular extensions are RoBERTa, DistilBERT and AlBERT.

There are some real benefits to using BERT. The most important one is that since it is pretrained, it can be used as input to a smaller model, avoiding recomputing the large BERT model. Moreover, if the fine-tuning is successful, a very good accuracy can be achieved. Also, BERT's performance depends on how big we want it to be. The BERT Large model has 340 million parameters and achieves much higher accuracies than the BERT Base which has 110 parameters. Furthermore, the Base version has twelve encoder layers, and the Large version twenty four. They both have large feedforward-networks (768 and 1024 hidden units, respectively), and many attention heads (12 and 16, respectively). Finally, pre-trained BERT models are available in more than a hundred languages. On the other hand, there is an

important drawback. Due to the fact that there is a lot of weights to update, it is slow to train, thus leading to the need of a lot of computing, so it is expensive.

2.2.5 BioBERT

The progress of NLP along with the increase of the number of biomedical literature has led to the growing development of expert, biomedical text mining for extracting information. However, even with the application of advanced NLP methodologies, such as Word2Vec and BERT, the performance of the models is not so good, due to the fact that those models are trained on texts of general context and not on expert, biomedical literature, where the corpora are different. Due to the fact that BERT achieves better and better performances, it is now adapted for the biomedical domain, so as to achieve more satisfactory results on biomedical texts. Thus, BioBERT was introduced (Bidirectional Encoder Representations from Transformers for Biomedical Text Mining), which is an NLP system focused and trained on biomedical corpora of a very large scale [30]. This pre-training has a drastic effect on the model's performance. More specifically, according to the evaluation of [30], BioBERT obtains higher F1 scores - F1 Score is the weighted average of *Precision* and *Recall*, score metrics that will be described in later chapters - in biomedical named entity recognition (NER) (0.62) and biomedical relation extraction (RE) (2.80), and a higher Mean Reciprocal Rank (MRR) score (12.24) in biomedical question answering (QA) than the current advanced models. In order to tackle the problem of words not existing in the vocabulary, BioBERT uses WordPiece tokenization, meaning that words are sub-tokenized to smaller tokens, which, added together, constitute the primary word.

BioBERT can be applied to several text mining tasks including *N*amed *E*ntity *R*ecognition (NER). NER is a technique used for identifying domain-specific nouns in a biomedical corpus. BioBERT learns WordPiece embeddings during pre-training and fine-tuning.

The technique of relation extraction is used for detecting relations of named entities in a biomedical corpus. Target named entities are anonymized in a sentence using pre-defined tags such as @GENE$ or @CHEMICALS$ [30].

Question answering is a technique within the fields of NLP in order to build systems that automatically answer questions posed in natural language given related passages. The fine-tuning of BioBERT for QA was done using the same BERT architecture used for SQuAD [31]. Token level probabilities for the start/end location of answer phrases were computed using a single output layer. Also, by using the same pre-training process of [32], the performance of both BERT and BioBERT was drastically improved.

While BERT often does not recognize some biomedical named entities, BioBERT will most probably recognize them and also find their exact boundaries. BioBERT can give correct answers to simple biomedical questions that BERT cannot, as well as provide longer named entities as answers.

2.3 Google Colab

Google Colab is an executable document that lets you write, run and share code with Google Drive [33]. It connects Notebook to a cloud-based runtime, thus giving the user the possibility to execute Python code without any required setup on their machine. If you would like to share your notebook with others, you can do so via Google Drive sharing or by exporting the Notebook to GitHub. Google Colab is essentially a Jupyter Notebook hosted on Google servers with additional functionalities, making it especially popular among data science artificial intelligence communities. So, the Notebooks created in Colab can be viewed and executed in Jupyter Notebook, JupyterLab and other compatible frameworks. Moreover, all the major libraries and tools are already pre-installed and pre-setup.

There are three types of servers that Google can allocate to the user: It is possible to choose among a GPU or a TPU hardware accelerator, or no accelerator, meaning that the code will run on CPUs. The accelerators are needed in cases that the algorithms need more computational power.

Google Colab gives us ~12-13 GB of RAM for free. However, most of BERT_Large's results cannot be reproduced using this amount of RAM memory, because the maximum batch size that can fit in memory is too small. That is why lighter BERT architectures are used, as mentioned in the previous section.

2.3.1 Jupyter

Jupyter is an open-source software that supports scientific computing and interactive data science across programming languages. It supports over 40 programming languages, such as Python and R [34].

Jupyter is centered around Jupyter Notebook, which is a web - based environment, where we can put text, mathematics and figures, explaining what we are going to do. Then, we can write the code that implements what we want to do. It is useful for researchers, as an interactive exploratory environment. Since it is a document-based interface, what you get in the end is a document that you can share.

Python is usually preferred because it is a language of choice for data science, for people who want to have "real" results. It combines the power of encrypting languages and the compiled languages have a very simple and easy to read syntax.

Chapter 3: Research methodology

As already mentioned, Google Colab offers free GPUs and TPUs. Since the computational power needed for this task is very big, we take advantage of this by attaching a GPU. The GPU was added by going to the menu and selecting: Edit → Notebook Settings → Hardware accelerator → (GPU).

Then the following code (Fig. 6) was run to confirm that the GPU was detected.

```
import tensorflow as tf

# Get the GPU device name.
device_name = tf.test.gpu_device_name()

# The device name should look like the following:
if device_name == '/device:GPU:0':
    print('Found GPU at: {}'.format(device_name))
else:
    raise SystemError('GPU device not found')
```

```
Found GPU at: /device:GPU:0
```

Fig. 6: GPU detection check.

In order for the torch library to use the GPU, the GPU was identified and specified as the device (Fig. 7). Later, data was loaded onto the device.

```
import torch

# If there's a GPU available...
if torch.cuda.is_available():

    # Tell PyTorch to use the GPU.
    device = torch.device("cuda")

    print('There are %d GPU(s) available.' % torch.cuda.device_count())

    print('We will use the GPU:', torch.cuda.get_device_name(0))

# If not...
else:
    print('No GPU available, using the CPU instead.')
    device = torch.device("cpu")
```

```
There are 1 GPU(s) available.
We will use the GPU: Tesla V100-SXM2-16GB
```

Fig. 7: In order for the torch library to use the GPU, the GPU was identified and specified as the device.

Afterwards, all the necessary for the implementation of our goal packages were imported, such as the library for regular expression operations, the BioC library in order to deal with BioCreative XML data, meaning to read from and write to it, the pandas library and others (Fig. 8).

```
import re
import os
import glob
import lxml.etree
import bioc
import json
from tqdm import tqdm
from neo4j import GraphDatabase
from collections import Counter, defaultdict
import pandas as pd
import csv
```

Fig. 8: Importation of necessary packages.

After the importation of the libraries, some basic functions were defined (Fig. 9). The *get_filenames* function seeks the path where all the BioC XML files are and returns those files. The *parse_bioc* function reads and returns the text from each file that it loads.

```
bioc_directory = 'E:\Λήψεις\output\BioCXML'

def now():
        return datetime.datetime.now()

def get_filenames():

    filenames_wild = os.path.join(bioc_directory, '*.xml')
    filenames = glob.glob(filenames_wild)

    print ('Found {} BioC XML files'.format(len(filenames)))
    #print(filenames)
    return filenames


def parse_bioc(filename):
    with open(filename, encoding="utf8") as f:
        text = f.read()
        try:
            b = bioc.loads(text)
        except lxml.etree.XMLSyntaxError as e:
            errors['could_not_parse'].append([filename])

    return b
```

Fig. 9: Definition of basic functions.

As already briefly mentioned above, the structure of the XML files is the following: Collection → Document → Passage → Annotation. *Collection* is the "title" of the file, the main/initial tag that encapsulates all the information of the file. The document tag corresponds to the papers. Thus, each document is one paper. The bio XML files that were used for this thesis include about 100 papers, each. The passage tag is what its name infers. Each passage corresponds to small passages of the whole text of each paper/document. Finally, the annotation tag encapsulates information about domain-specific nouns, such as their name (e.g., GLUT9) and their type (e.g., *Gene*).

```
def parse_collection(collection):
    delme = defaultdict(int)
    #wholetext=defaultdict(dict)
    wholetext = set()
    specifictext_D=set()
    specifictext_S=set()
    specifictext_G=set()
    specifictext_C=set()
    c = 0

    #for document in collection['documents']:
    for document in collection.documents:
        genes = set()
        for passage in document.passages:
            text = passage.text
            wholetext.add(text)
            #print (text)

            for annotation in passage.annotations:

                delme[annotation.infons['type']] += 1
                c += 1

                if c > 10000:
                    pass

                if annotation.infons['type'] == 'Disease':
                    specifictext_D.add(annotation.text)

                elif annotation.infons['type'] == 'SNP':
                    specifictext_S.add(annotation.text)

                elif annotation.infons['type'] == 'Gene':
                    specifictext_G.add(annotation.text)

                elif annotation.infons['type'] == 'Chemical':
                    specifictext_C.add(annotation.text)

    return wholetext, specifictext_D, \
           specifictext_S, specifictext_G, specifictext_C
```

Fig. 10: Definition of basic functions.

Having mentioned all these, the *parse_collection* function scans each paper and returns the whole text of the file and then all the entities (Fig. 10). More specifically, four sets were created, one containing all the *Disease* entities, one containing all the *Gene* entities, one containing all the *Chemical* entities and finally, the fourth set contains all the *SNP* entities.

The *import_graph* function calls the *get_filenames* function and the *parse_bioc* function and returns the whole text and all the entities from all the XML files that exist in the given directory, again separated into their own sets (Fig. 11).

```
def import_graph():
    filenames = get_filenames()
    wtext=set()
    stext_D=set()
    stext_S=set()
    stext_G=set()
    stext_C=set()
    for filename in filenames:
        collection = parse_bioc(filename)
        wtext.update(parse_collection(collection)[0])
        stext_D.update(parse_collection(collection)[1])
        stext_S.update(parse_collection(collection)[2])
        stext_G.update(parse_collection(collection)[3])
        stext_C.update(parse_collection(collection)[4])

    return wtext, stext_D, stext_S, stext_G, stext_C
```

Fig. 11: Definition of basic functions.

Here, the *import_graph* function is called, so the *wtext* variable contains the whole text and the variables *stext_D*, *stext_S*, *stext_G*, *stext_C* contain the *Disease*, the *SNP*, the *Gene* and the *Chemical* entities, respectively. For this study, 1,000 files were parsed, which are translated to roughly 100,000 papers.

The *wtext* variable was converted to list and the *list_wt_v* variable made some changes to specific sentences that lacked some spaces or a full stop, so as to be in the correct form for a future part of the algorithm (Fig. 12).

The *datatostring* function converts the given list (the *list_wt_v* list in our case) of sentences to strings (Fig. 12).

The next step is depicted in Fig. 13. Regular expressions were used, in order to split the sentences by the dot (“.”) delimiter. Then the algorithm scanned the sentences one by one and each sentence was split to its words. After this “tokenization”, the algorithm, having the sets with the entities, searched

```
list_wt_v = list(filter(None, list(wtext)))
for i in range(0, len(list_wt_v)):
    if list_wt_v[i][-1] != '.' :
        list_wt_v[i] = ' ' + list_wt_v[i] + '. '
def datatostring(list_wt_v) :
    my_str_wt = ''
    for x in list_wt_v :
        my_str_wt += '' + x

    return my_str_wt
```

Fig. 12: Conversion of the *list_wt_v* variable to the correct form for a future part of the algorithm.

each token and if it identified an entity, it would add the sentence that has this token to a new set, called *entities*.

Otherwise, the rest of the sentences were added to another set, called *set_wd*. This set has all the sentences that do not contain any entity at all. After that, each sentence of the set *entities* was

```
div_wt = re.split(r'(?<=[^A-Z].[.?]) +(?=[A-Z])', datatostring(list_wt_v))
div_wt = set(div_wt)
print("Whole text length:", len(div_wt))
print('')

disease=set()
snp=set()
gene=set()
chemical=set()
entities=set()
set_wt=set()

for k in div_wt:
    a = set(k.split())
    for y in a:
        if y in stext_D or y in stext_S or y in stext_G or y in stext_C:
            entities.add(k)
            break
    else:
        ######## no entity at all S3(E) #########
        set_wt.add(k)

####### has entity in general S1(E) #########
for k in entities:
    a = set(k.split())
    for y in a:
        if y in stext_D:
            disease.add(k)
        elif y in stext_S:
            snp.add(k)
        elif y in stext_G:
            gene.add(k)
        elif y in stext_C:
            chemical.add(k)
```

Fig. 13: Creation of sets containing different entities (part 1).

scanned and “tokenized”. The algorithm deciphered the type of the entity that the scanned sentence contained and put this sentence to a new set that only contains this kind of sentences. Thus, if the algorithm identifies an entity which is a *Disease*, it adds the corresponding sentence to set that only includes sentences which have at least one *Disease* entity. The same applies for the rest three entities. It is important to mention that these new sets do not include sentences that only have one type of entity. For example, one sentence can contain both a *Disease* and a *Gene*. This means that the same sentence will be added to two sets.

The next step was to create sets that contain sentences with only one entity each and sets that contain any entity but one, each (see Fig. 14). For instance, the set named *not_disease* contains sentences that include *Genes*, *SNPs* or *Chemicals* but no *Disease* entities.

```
#########   only entities S4(E)   ##########
disease_only=disease-snp-gene-chemical
snp_only=snp-disease-gene-chemical
gene_only=gene-disease-snp-chemical
chemical_only=chemical-disease-snp-gene

#########   no E but any other entity S2(E)   ##########
not_disease=set.union(snp,gene,chemical)-disease
not_snp=set.union(disease,gene,chemical)-snp
not_gene=set.union(disease,snp,chemical)-gene
not_chemical=set.union(disease,snp,gene)-chemical

print("disease text length:", len(disease))
print("snp text length:", len(snp))
print("gene text length:", len(gene))
print("chemical text length:", len(chemical))
print('')

print("not disease text length:", len(not_disease))
print("not snp text length:", len(not_snp))
print("not gene text length:", len(not_gene))
print("not chemical text length:", len(not_chemical))
print('')

print("No entities text length:", len(set_wt))
print('')

print("only disease text length:", len(disease_only))
print("only snp text length:", len(snp_only))
print("only gene text length:", len(gene_only))
print("only chemical text length:", len(chemical_only))
```

Fig. 14: Creation of sets containing different entities (part 2).

In table 1, the number of sentences of each aforementioned set is depicted.

Table 1: Number of sentences per entity and category.

Total sentences	7,995,701		
Sentences with no entities	299,718		
	Sentences that:		
	have this entity in general	do not have this entity but have other entities	have only this entity
Disease	3,353,854	4,422,129	33,574
SNP	3,432	7,769,516	22
Gene	7,526,112	249,871	966,635
Chemical	6,317,236	1,458,747	158,584

At the final step of this pre-processing part of the code, meaning the preparation of the dataset, two datasets were created for each entity: More specifically, the first dataset, which was named *ML12*, contained the sentences that had a specific entity and the sentences that contained any other entity except for this specific one. The second dataset, which was named *ML13*, again contained the sentences that had a specific entity and sentences that did not contain any entity at all. For each dataset, one category of sentences was labeled with the number 0 and the other with the number 1, so as to be separated. In total, eight datasets were created, two for each entity (*Disease*, *Gene*, *SNP*, *Chemical*).

The code below depicts the procedure followed, in order to create the two datasets for the case of the *Disease* entity (see Fig. 15,16). At first, the sentences of the already calculated set that contains *Disease* entities were labeled with 1, whereas the sentences that contained any other entity were labeled with 0. Finally, all datasets were extracted in a .tsv as well as in a .csv format for the needs of the following steps. The same procedure was followed for the rest of the entities, so in total, eight .tsv and eight .csv files were created.

It is important to mention that only sentences that were comprised of more than 15 and less than 300 letters were extracted. As it was observed, most sentences with less than 15 letters corresponded to titles of paragraphs, or the declaration of the authors of the papers, that is why they were excluded. The reason why sentences with more than 300 letters were excluded had to do with the computational power during the Machine Learning process that will be described in a more analytic way, in the next Chapter.

```
######### DISEASE #########
###### ML12(E) #######
final_disease=list(disease) #label-->1
final_not_disease=list(not_disease) #label-->0

#creating labels
lab_nd=[]
for lab1 in range(0, len(final_not_disease)):
    lab_nd.append(0)

lab_d=[]
for lab2 in range(0, len(final_disease)):
    lab_d.append(1)

#append the length of each element to list
new_d=[]
for t1 in range(0,len(final_disease)) :
    d1=len(final_disease[t1])
    new_d.append(d1)

new_nd=[]
for t2 in range(0,len(final_not_disease)) :
    nd2=len(final_not_disease[t2])
    new_nd.append(nd2)

#creating the final file
df_d_1 = pd.DataFrame({
    'Sentences': final_disease,
    'Labels': lab_d,
    'Length': new_d } )

df_d_1 = df_d_1[df_d_1['Length'] > 15]
df_d_1 = df_d_1[df_d_1['Length'] < 300]
df_d_1 = df_d_1.reset_index(drop=True)
#df_d_1 = df_d_1.iloc[50:100]

df_d_2 = pd.DataFrame({
    'Sentences': final_not_disease,
    'Labels': lab_nd,
    'Length': new_nd } )
```

Fig. 15: Dataset extraction (part 1).

Up to this point of the code, the model was running for about two hours. The part where the processing of the algorithm “stack” was the parsing of the BioC files, due to their high volume.

The next and final step was the classification of the sentences containing those four biomedical entities, so as to answer the question about which types of entities can be easily recognized. The aim of the model is, after being given a sentence, just like those given in the dataset, to be able to label it with either 1 or 0, thus informing the user if it contains a specific biomedical entity (*SNP*, *Disease*, *Gene*

```
df_d_2 = df_d_2[df_d_2['Length'] > 15]
df_d_2 = df_d_2[df_d_2['Length'] < 300]
#df_d_2 = df_d_2.iloc[:50]

df_d_a=df_d_1.append(df_d_2)
df_d_a=df_d_a[['Sentences','Labels']]
df_d_a.to_csv('test_label_d1.csv',index=False, header=False)

csv.writer(open("disease_ML12.tsv", 'w+'), delimiter='\t').\
writerows(csv.reader(open("test_label_d1.csv")))

###### ML13(E) #######
final_no_entity=list(set_wt) #label-->0

#creating labels
lab_ne=[]
for lab1 in range(0, len(final_no_entity)):
    lab_ne.append(0)

new_ne=[]
for t2 in range(0,len(final_no_entity)) :
    ne2=len(final_no_entity[t2])
    new_ne.append(ne2)

#creating the final file
df_d_3 = pd.DataFrame({
    'Sentences': final_no_entity,
    'Labels': lab_ne,
    'Length': new_ne } )

df_d_3 = df_d_3[df_d_3['Length'] > 15]
df_d_3 = df_d_3[df_d_3['Length'] < 300]
#df_d_3 = df_d_3.iloc[:5]

df_d_final_b=df_d_1.append(df_d_3)
df_d_final_b=df_d_final_b[['Sentences','Labels']]
df_d_final_b.to_csv('test_label_d2.csv',index=False, header=False)

csv.writer(open("disease_ML13.tsv", 'w+'), delimiter='\t').\
writerows(csv.reader(open("test_label_d2.csv")))
```

Fig. 16: Dataset extraction (part 2).

or *Chemical*), if it contains entities except for a specific one, or finally, if it doesn't contain any biomedical entity at all.

To begin with, the huggingface Transformers library [3,4] was installed, in order for the NLP model to be able to be loaded (Fig. 17).

```
!pip install transformers
```

Fig. 17: Installation of the huggingface Transformers library.

Then, all the necessary tools were imported (Fig. 18).

```
import numpy as np
import numpy
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
import sklearn.linear_model as linear_model
import sklearn.metrics as metrics
from sklearn.metrics import confusion_matrix
from sklearn.dummy import DummyClassifier
import torch
import transformers as ppb
from sklearn.utils import shuffle
import warnings
warnings.filterwarnings('ignore')
```

Fig. 18: Importation of necessary libraries.

After that, Google Drive was mounted to Google Colab. All the dataset files that were extracted before, were uploaded to Google Drive (Fig. 19).

```
from google.colab import drive
drive.mount('/content/gdrive')
```

Fig. 19: Mount of Google Drive to Google Colab.

By mounting Google Drive to Google Colab, the latter had access to the dataset files, so they were just imported one by one directly into a pandas dataframe (Fig. 20).

```
df = pd.read_csv('/content/gdrive/MyDrive/Colab Notebooks/labels_new \
                  /gene_ML12.tsv', delimiter='\t', header=None)
```

Fig. 20: Dataset importation.

Each dataset file was sorted by the label value, meaning that all sentences labeled with 0 were output after all the sentences labeled with 1. This means that if I want to pick a subset of the whole pandas dataframe, let's say the first 2,000 sentences, in order to proceed with the classification, only sentences labeled with 1 will be picked. That is why the dataframe was firstly shuffled (Fig. 21).

```
df = shuffle(df)
```

Fig. 21: Dataset shuffle.

By executing the code depicted below, the number of "0s" and "1s" were printed. For the case of *Genes* and the ML12 dataset for example, we can see that the "1s" are much more than the "0s" (see Fig. 22).

```
df[1].value_counts()
```

```
1    2823453
0     216910
Name: 1, dtype: int64
```

Fig. 22: Count of sentences with label 0 and sentences with label 1.

If we choose to run the classification with (the first) 2,000 sentences in total for example, we observe that the "1s" are again much more than the "0s" (Fig. 23).

```
batch_1 = df[0:2000]
batch_1[1].value_counts()
```

```
1    1862
0     138
Name: 1, dtype: int64
```

Fig. 23: Number of sentences labeled with 0 and 1 after choosing the first 2,000 sentences before the adjustment of the dataset range.

That is why the dataset was modified and adjusted appropriately for each entity, for each case, so as to have a balanced number of sentences, independently of the chosen range (Fig. 24).

```
df = df[2606543:3040363]
batch_1 = df[0:2000]
batch_1[1].value_counts()
```

```
1    1008
0     992
Name: 1, dtype: int64
```

Fig. 24: Number of sentences labeled with 0 and 1 after choosing the first 2,000 sentences after the adjustment of the dataset range.

This algorithm was split into two models, DistilBERT and Basic Logistic Regression.

As already mentioned, DistilBERT is a more compact extension of BERT architecture, developed by the Hugging Face team [3,4]. Hugging Face team applied a technique called distillation, which compresses a large model (teacher) into a smaller model (student), and knowledge distillation, where the student is trained to reproduce the behavior of the teacher model. For our case, the student is a small version of BERT, without the token-type embeddings and the pooler that is used for the next sentence classification. The layers were reduced to half and the rest of the architecture stayed the same. Despite the aforementioned changes, the hidden size was kept constant, since changes in the hidden dimension wouldn't affect the Transformer architecture. So, reducing the hidden size would reduce the parameters and make the model smaller. However, it wouldn't make it faster.

The training setup is purposely limited in terms of resources. DistilBERT is trained on eight 16GB V100 GPUs for approximately three and a half days, using the concatenation of Toronto Book Corpus and English Wikipedia (same data as original BERT) [3].

Deep learning inference is the process of using a trained DNN model to make predictions against previously unseen data. So, as far as inference time is concerned, distilBERT is 60% faster than BERT.

All in all, distilBERT reduces the size of a BERT model by 40%, while retaining 97% of its language understanding capabilities and being 60% faster.

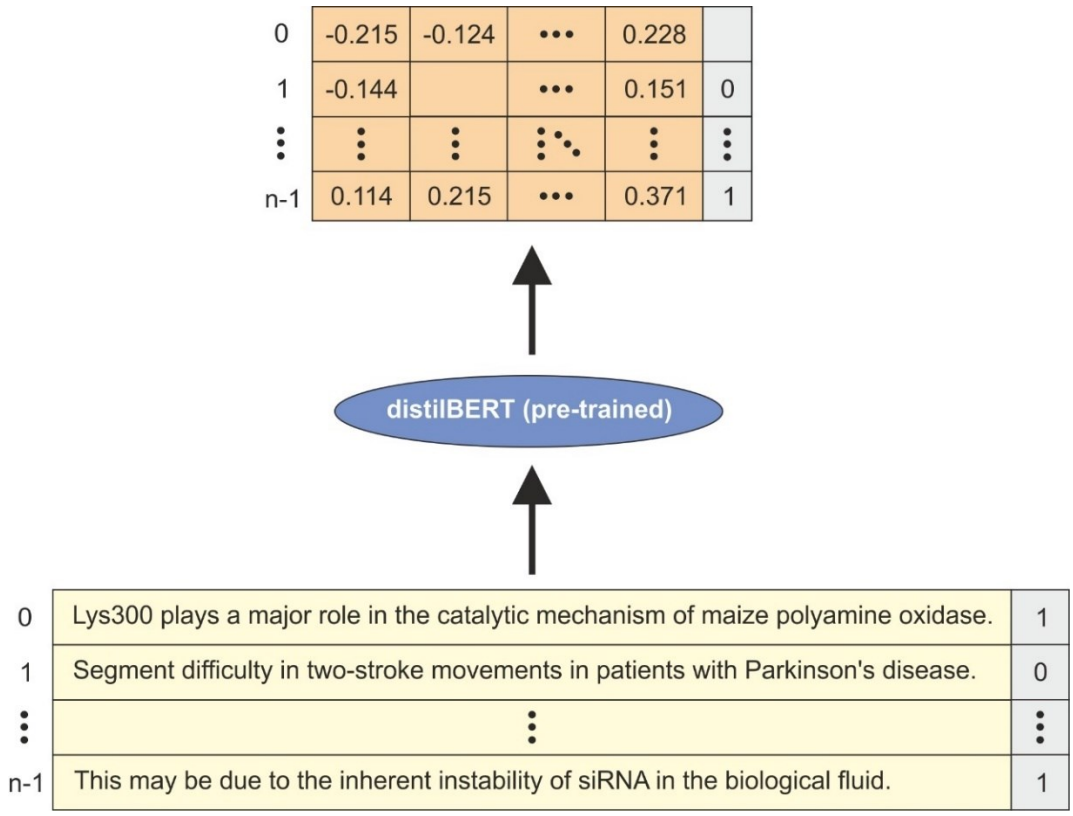


Fig. 25: Creation of the embeddings for the input sentences using the pre-trained distilBERT model.

The pre-trained distilBERT was used, so as to create the embeddings for the sentences initially given from the dataset as input (Fig. 25).

After the distilBERT's processing was done, this extracted output was taken by a basic Logistic Regression model from scikit-learn Machine Learning library [35]. The role of this basic Logistic Regression model was to classify the input sentence to either 1 or 0. Logistic Regression is a statistical method for predicting two or more binary classes [36]. It is based on the sigmoid function (Fig. 26) and the natural logarithm function (Fig. 27).

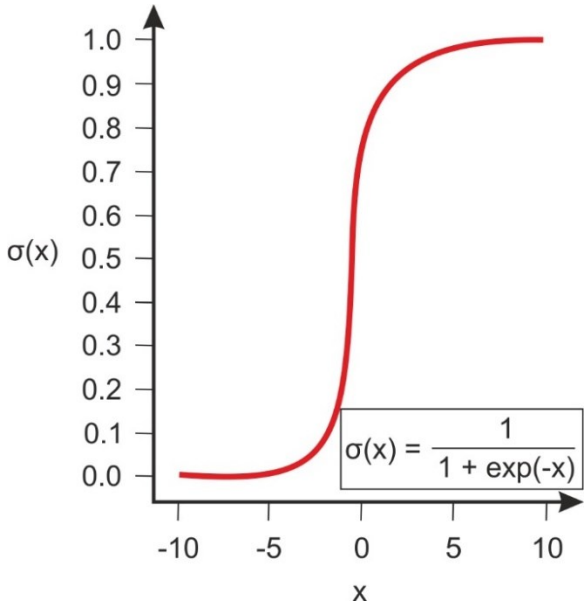


Fig. 26: Sigmoid function curve.

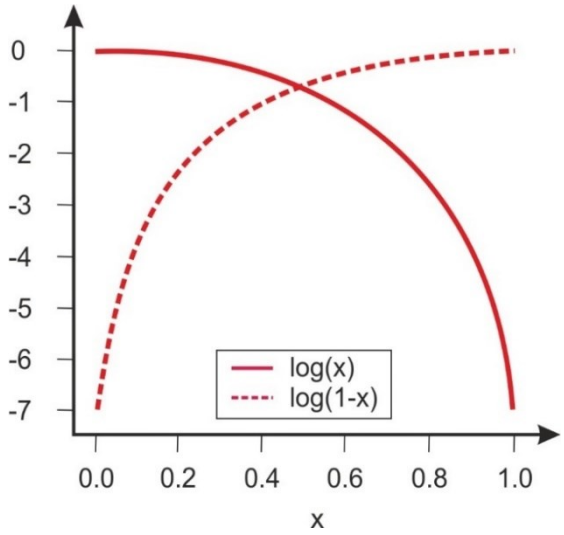


Fig. 27: Natural logarithm function curve.

The sigmoid function is applied for classification methods, due to the fact that it has values close to 0 and 1 across most of its domain. Fig. 27 depicts the natural logarithm log(x) of some variable x, for values of x between 0 and 1.

Let's say that we have a set of r independent variables $\mathbf{x}$ = (x_1, …, x_r) and we want to implement the Logistic Regression of a dependent variable y. We start with the known variables of the x_i inputs and the corresponding output y_i for each observation i = 1, …, n. The aim is to find the Logistic Regression function p(x) such that the predicted outputs $p(\mathbf{x}_i)$ are as close as possible to the actual output y_i for each observation i = 1, …, n. Thus, for our case, as close as possible to either 0 or 1. That is why the sigmoid function is used.

Having the Logistic Regression function $p(\mathbf{x})$, we can predict the outputs for new and unseen inputs.

The basic methodology of Logistic Regression is the following. Since it is a linear classifier, the linear function $f(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$ is used. The variables b_i are called the predicted weights or coefficients. The Logistic Regression function $p(\mathbf{x})$ is the sigmoid function of $f(\mathbf{x})$: $p(\mathbf{x}) = 1 / (1 + \exp(-f(\mathbf{x}))$. That is why, it is close to either 0 or 1. The function $p(\mathbf{x})$ is the predicted probability that the output for a given $\mathbf{x}$ is equal to 1 and $1 - p(x)$ is the probability that the output is 0. Logistic Regression determines the best predicted weights such that the function $p(\mathbf{x})$ is as close as possible to all actual outputs y_i, i = 1, …, n, where n is the number of observations. The process of calculating the best weights using available observations is called model training or fitting.

Fig. 28 depicts a general view of the whole process described above. The data passed between these models is a vector of size 768 (BERT Base), which is an embedding for the input sentence and comes from the result of the first position, which receives the [CLS] token as input.

Between the above two models, only the Logistic Regression model was trained. For distillBERT, an already pre-trained model was used. However, it was trained or fine-tuned for sentence classification.

Thus, the output of distilBERT was taken and split into a training and a testing set, so as for Logistic Regression to evaluate this new dataset (Fig. 29). By default, 75% of the whole set was used as the training set and the rest 25% was used as the testing set. Of note, the output was first shuffled before being split, meaning that it doesn't just take the first 75% of examples as they appear in the dataset.

The next step was to train the Logistic Regression model on the training set.

Let's go back to our code. A pre-trained distilBERT model was loaded with a distilBERT tokenizer (Fig. 30). The function loaded all the pre-trained weights for us from the distilBERT layers. The *distilbert-base-uncased* model is a distilBERT model, pre-trained on the same data used to pre-train BERT, using

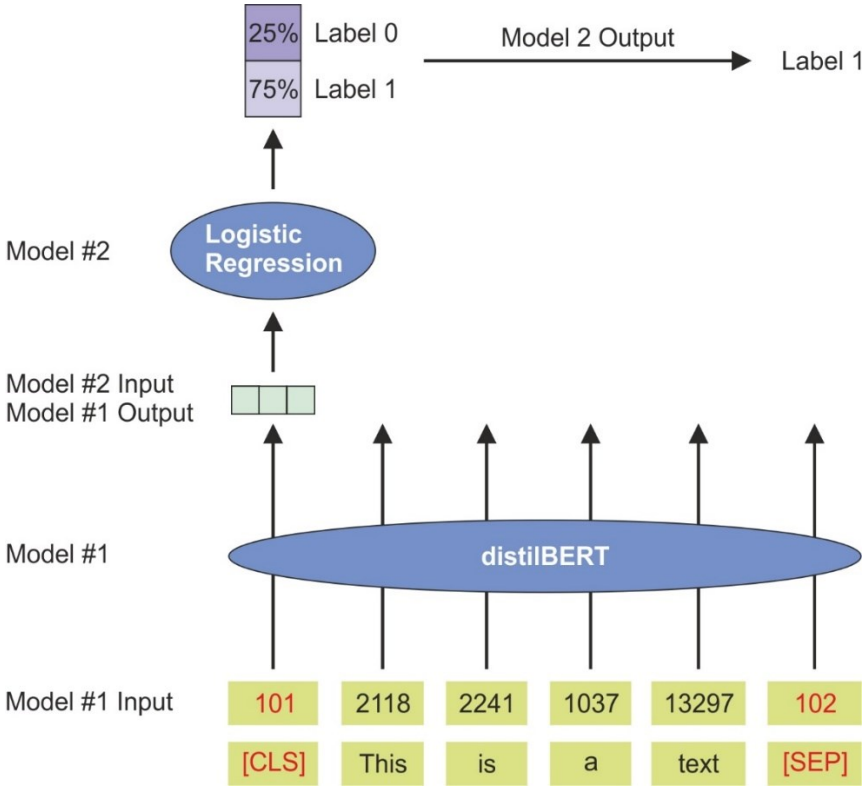


Fig. 28: A general depiction of the flow path through distilBERT till the output of the label.

distillation with the supervision of the "bert-base-uncased" version of BERT. "bert-base-uncased" means that it is the version that has only lowercase letters and is the smaller ("base") version of the

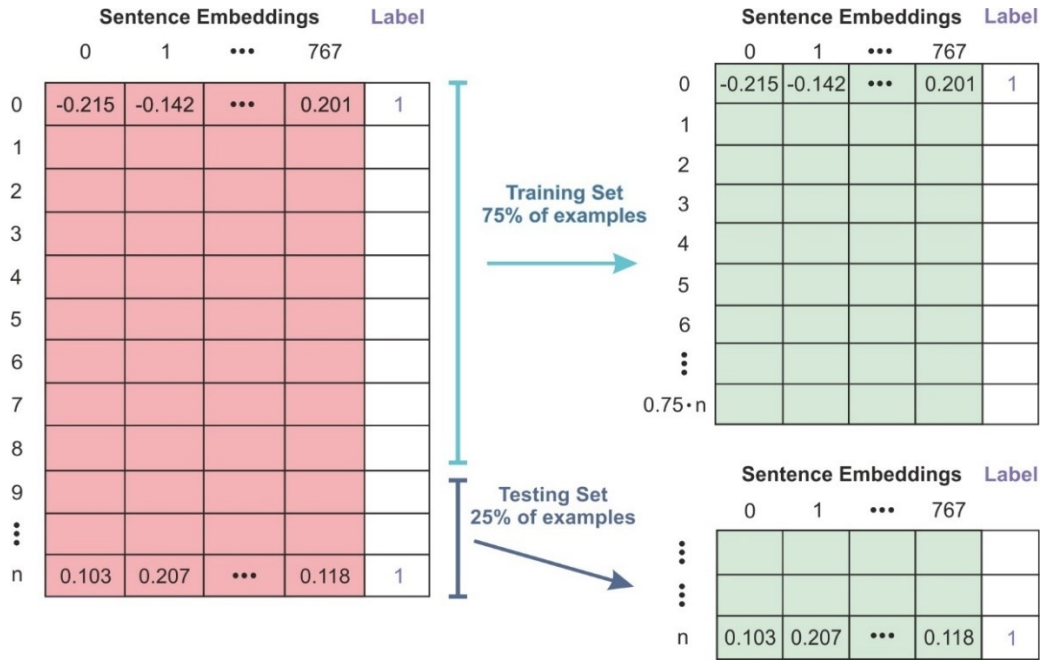


Fig. 29: Train/Test split for the output of distilBert model creates the dataset that will be trained and evaluated on the Logistic Regression model.

two. The model has 6 layers, 768 dimensions and 12 layers/heads, totalizing 66M parameters. In BERT uncased the text has been lowercased before the WordPiece tokenization step.

The pre-trained tokenizer was loaded based on the *"distilbert-base-uncased"* model, with a vocabulary size of 30,522 words, a maximum sentence length of 512 tokens and the following special tokens: 'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]' [4].

In order for the sentences to be passed through BERT, they need to be pre-processed, meaning to be adjusted such that their format is compatible with BERT's requirements.

So, the first step was the tokenization of the sentences. In other words, each sentence was split into words and subwords (Fig. 30). It is important to mention again that BioBERT does not have its own vocabulary of tokens based on the corpus.

```
# DistilBERT:
model_class, tokenizer_class, pretrained_weights = (ppb.DistilBertModel, \
                      ppb.DistilBertTokenizer, 'distilbert-base-uncased')

# Load pretrained model/tokenizer
tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
model = model_class.from_pretrained(pretrained_weights)
```

Fig. 30: Pre-trained distilBERT model loaded with a distilBERT tokenizer.

Instead, it uses the same tokens as the original BERT, so as to maintain compatibility. The tokenization was done for all sentences together as a batch (Fig. 31).

```
tokenized = batch_1[0].apply((lambda x:\
tokenizer.encode(x, add_special_tokens=True)))
```

Fig. 31: Tokenization for all sentences together as a batch.

After tokenization, each sentence was turned into a list of numbers, of IDs (Fig. 32).

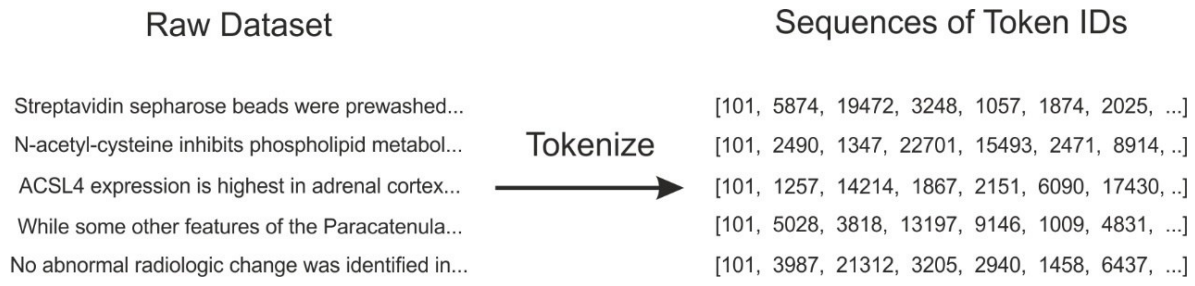


Fig. 32: Conversion of sentences to lists of numbers.

The next step of pre-processing was padding. All the lists were padded to the same size, so the vectors were made the same size, by adding the 0 token to the shorter sentences, till their length reached the length of the longest sentence (Fig. 33).

In case that some sentences' length is longer than 512 tokens, truncation is necessary as well. That means the opposite of padding. Some tokens need to be dropped, hoping that the remaining text is enough to perform the task well.

```
max_len = 0
for i in tokenized.values:
    if len(i) > max_len:
        max_len = len(i)

padded = np.array([i + [0]*(max_len-len(i)) for i in tokenized.values])
```

Fig. 33: Padding of lists to the same size.

The new matrix consists of many zero values due to the padding. Those 0 padding tokens are incorporated into the model's decision making, impacting both the training speed and the test set accuracy. That is why an array was created in order to indicate the padding and ignore it (Fig. 34).

```
attention_mask = np.where(padded != 0, 1, 0)
```

Fig. 34: Creation of the attention mask.

The new matrix is now ready to be passed to distilBERT (Fig. 35).

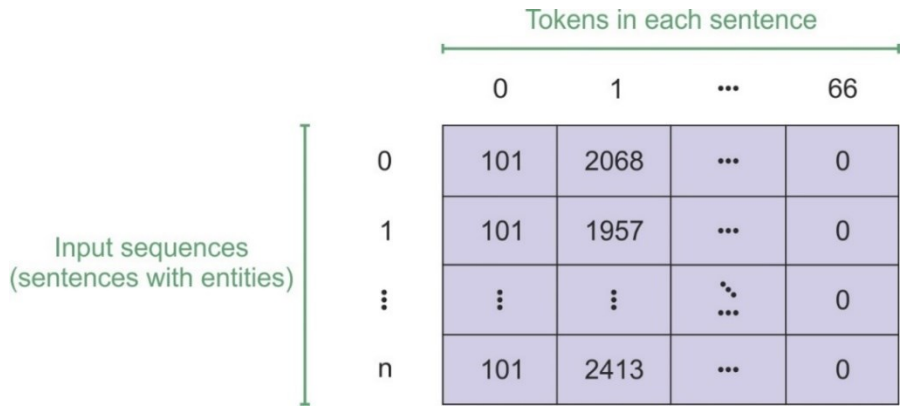


Fig. 35: The final matrix that is passed to distilBERT. This matrix is the result of the *print(padded)* command. It depicts the tokens in each sentence of n input sentences.

Then, the input tensor and the attention mask arrays were created via the padded token matrix and the attention mask array, respectively, in order to be sent to distilBERT (see Fig. 36).

```
input_ids = torch.tensor(padded)
attention_mask = torch.tensor(attention_mask)

with torch.no_grad():
    last_hidden_states = model(input_ids, attention_mask=attention_mask)

last_hidden_states[0].shape
```

```
torch.Size([2000, 59, 768])
```

Fig. 36: Creation of the input tensor and the attention mask arrays.

This step was run with *torch.no_grad()*. *Torch.no_grad()* is a context manager that deactivates gradient calculation. It contributes to the reduction of the memory usage and it speeds up computations.

The variable *last_hidden_states* holds the results of the processing. It is a tuple with shape (number of sentences) x (max number of tokens of the longest sentence) x (768 which is the number of hidden units in the distilBERT model) (Fig. 36,37).

As we know, BERT consists of 12 Transformer layers. On the output of the 12th Transformer, only the first embedding is used by the classifier. The [CLS] token, which corresponds to the first token of each sentence, is the only significant part of the sentence when we apply sentence classification. That is why the rest tokens of the sentences are discarded and only the output for the [CLS] token is kept, which represents an embedding for the entire sentence.

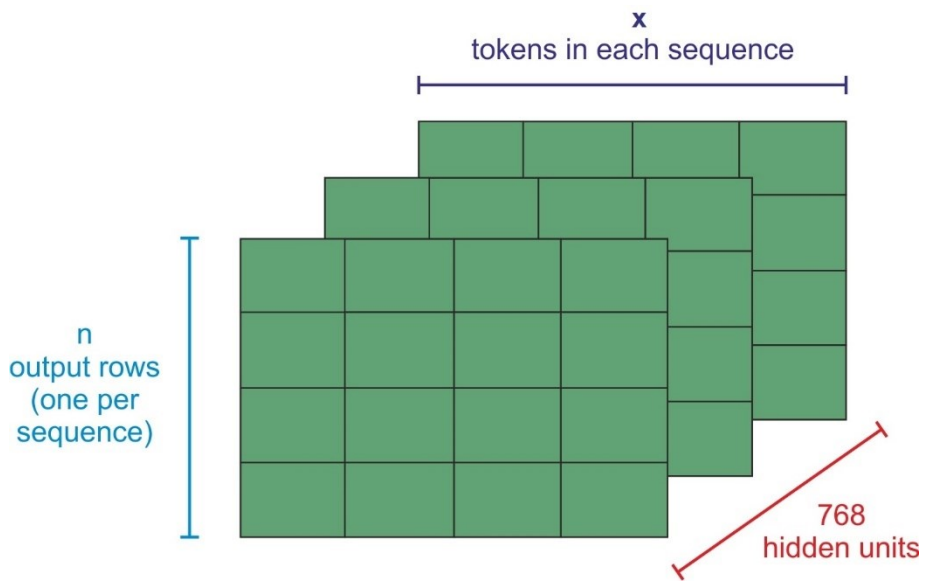


Fig. 37: The optical output of the *last_hidden_states* variable. BERT output tensor / predictions.

The 3D *last_hidden_states* matrix becomes now a 2D matrix called features, after applying the following command (Fig. 38).

```
features = last_hidden_states[0][:,0,:].numpy()
features.shape
```

```
(2000, 768)
```

Fig. 38: The new 2D *last_hidden_states* matrix and its shape.

Each row corresponds to a sentence of the input dataset. Each column corresponds to the output of a hidden unit from the FFNN (see Fig. 39).

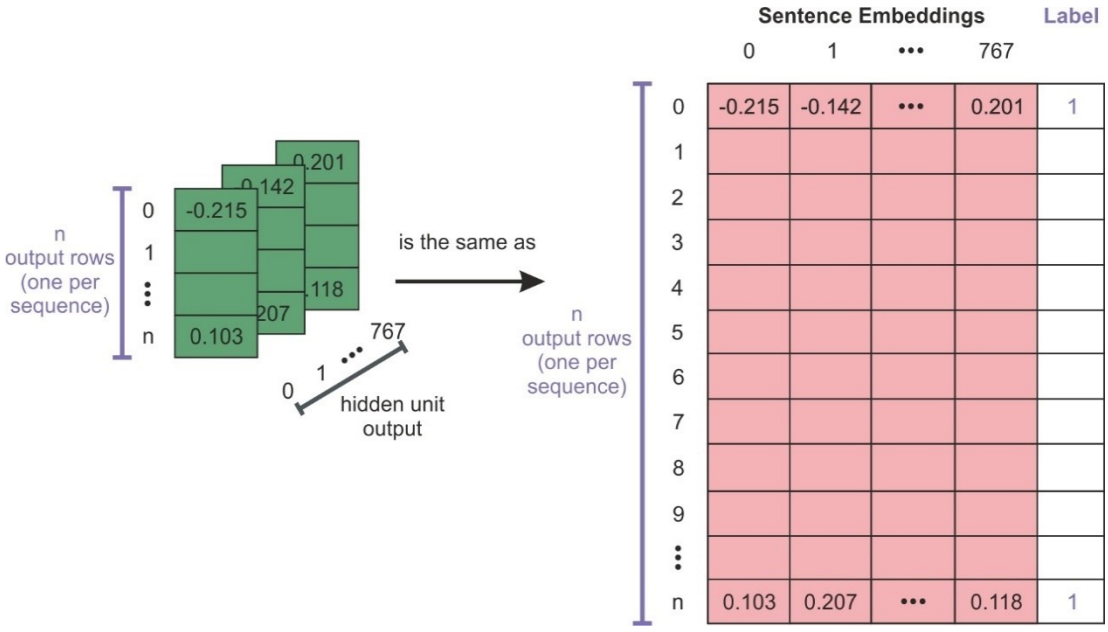


Fig. 39: A 2D numpy array containing the sentence embeddings of all the (n) sentences in the dataset.

Now, only the values of the labels from variable batch_1 into the variable labels were kept (Fig. 40).

```
labels = batch_1[1]
```

Fig. 40: Assignment of all labels to a new variable.

Then, our dataset was split into a training set and testing set (75% - 25% by default). So, now we have got our dataset divided into training inputs and validation inputs, along with their labels and their attention masks.

In the past, the data was shuffled and split by hand, using pandas. However, all this procedure does not need to be done, since scikit-learn has consolidated these tasks into one function, the

train_test_split function (Fig. 41). The *train_test_split* function creates training and test data for the features and the labels.

```
train_features, test_features, train_labels, test_labels = \
train_test_split(features, labels)
```

Fig. 41: Creation of the training and test data for the features and the labels.

3.1 Calculation of a prediction

In a nutshell, the steps needed in order to reach a prediction of the output, given a sentence as input, provided that the model is trained, are the following:

- The BERT tokenizer splits the word into tokens.
- The special tokens needed for sentence classification, [CLS] at the first position and [SEP] at the end of the sentence, are added.
- Each token is replaced with its embedding thanks to the trained model.

Note that the tokenizer does all these steps in a single line of code:

tokenizer.encode("a visually stunning rumination on love", add_special_tokens=True)

- The input sentence is then passed to distilBERT which gives as output a vector for each input token. This vector contains 768 numbers for each token.
- Since this task has to do with sentence classification, only the first vector (the one associated with the [CLS] token) is taken into account and passed into the Logistic Regression model. The Logistic Regression model by its turn, classifies the vector thanks to the knowledge it gained from its training phase.

Chapter 4: Research findings / results

The Logistic Regression model was trained on the training set (Fig. 42). So, now the machine is officially learning and is ready to make predictions on unseen data.

The method "fit" trains the model. Model fitting is the process of determining the coefficients b_0, b_1, …, b_r that correspond to the best value of the cost function.

```
model = LogisticRegression()
model.fit(train_features, train_labels)
```

Fig. 42: Training of the Logistic Regression model.

The obtained string representation of the fitted model is as follows (Fig. 43).

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

Fig. 43: String representation of the fitted model.

We observe several parameters which are responsible for the behavior of the model and which can be adjusted. Some parameters worth mentioning are:

- C, which is a positive floating-point number that defines the relative strength of regularization. Smaller values indicate stronger regularization. The default number 1 was applied in our case.
- class_weight, which is a dictionary that defines the weights related to each class. The default 'None' means that all classes have the weight 1.
- max_iter, which is an integer that defines the maximum number of iterations by the solver during model fitting. The default input is 100.
- solver is a string ("liblinear" by default) that decides which solver to use for the model fitting. The default option that was used is "liblinear". This option is recommended when you have high dimension dataset (recommended for solving large-scale classification problems). Other options are "newton-cg", "lbfgs", "sag", and "saga". More information about solvers can be found in [37].

After the end of the training, evaluation was done. In order to get a holistic view of how well the model performed, the confusion matrix was extracted [38] (see Fig. 44). A confusion matrix is used for evaluating the performance of a classification model. It compares the actual target values with those predicted by the Machine Learning model.

```
import sklearn.linear_model as linear_model
import sklearn.metrics as metrics
import numpy
from sklearn.metrics import confusion_matrix

y_pred = model.predict(test_features)

cf = confusion_matrix(test_labels, y_pred)
print('confusion matrix: \n', cf)

accuracy = (cf[0,0] + cf[1,1]) / (cf[0,0] + cf[1,0] + cf[0,1] + cf[1,1])
print('accuracy: ', accuracy)

precision = (cf[0,0]) / (cf[0,0] + cf[0,1])
print('precision: ', precision)

recall = cf[0,0] / (cf[0,0] + cf[1,0])
print('recall: ', recall)
```

```
confusion matrix:
 [[204  47]
 [ 45 204]]
accuracy:  0.816
precision:  0.8127490039840638
recall:  0.8192771084337349
```

Fig. 44: Evaluation of the model.

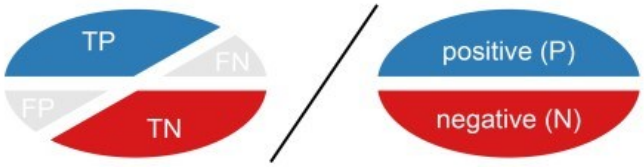
In our case (binary classification), the confusion matrix shows the numbers of the following:

- True positives in the upper-left position
- False positives in the upper-right position
- False negatives in the lower-left position
- True negatives in the lower-right position

The columns represent the actual values of the target variable and the rows represent the predicted values of the target variable. True Positive means that the actual value was positive and the model predicted a positive value. False Positive means that the actual value was negative but the model predicted a positive value. False Negative means that the actual value was positive but the model predicted a negative value. Finally, True Negative means that the actual value was negative and the model predicted a negative value.

The computation of the confusion matrix allows more detailed analysis than mere proportion of correct classifications (*Accuracy*). Sometimes, *Accuracy* is giving the wrong idea about the results when the numbers of observations in different classes vary greatly. For example, if there are 90 sentences that contain *Gene* entities and 10 that contain no entities in the data, the classifier might

classify all the observations as sentences with *Gene* entities. *Accuracy* is the ratio of the number of correct predictions to the total number of predictions (or observations) (Fig. 45).



$$\text{Accuracy} = \frac{TP+TN}{TP+FP+FN+FP}$$

Fig. 45: The Accuracy metric.

Thus, although the *Accuracy* is 95%, the classifier has 100% sensitivity for the sentences with the *Genes* and will not recognize any of the other types of sentences.

That is why, two more concepts were used in order to extract a better evaluation, as they are more informative in cases when one wants to avoid false negatives more than false positives or vice versa, *Precision* and *Recall*.

$$\text{Precision} = \frac{TP}{TP+FP}$$

Fig. 46: The *Precision* metric.

Precision indicates how many of the correctly predicted cases are actually positive and determines the reliability of the model (Fig. 46). It is especially useful when False Positives are more important than False Negatives.

Recall indicates how many of the actual positive cases are correctly predicted by the model (Fig. 47). It is especially useful when False Negative is more important than False Positive. *Recall* is important in medical cases where the actual positive cases should always be detected and predicting False Negative is not harmful.

$$\text{Recall} = \frac{TP}{TP+FN}$$

Fig. 47: The *Recall* metric.

As already mentioned, the code described above was applied in order to perform two different tasks.

The first task, ML12 task, had to do with the classification between sentences that contain at least one specific type of entity among *Disease*, *Gene*, *SNP* and *Chemical*, and sentences that contain at least one of the rest three entities but not the given entity.

For example, for the case of *Genes*, a group of sentences contain at least one *Gene*, each. This means that one sentence can contain other types of entities as well or no other entity. The other group of sentences, labeled with 0, does not contain a *Gene*, but it must contain any other of the rest three entities (*Chemical*, *SNP*, or *Disease*).

The second task, ML13 task, had to do with the classification between sentences that again contain at least one specific type of entity, but the other group of sentences, labeled with 0, does not contain any entity at all.

The initial goal of this study was to perform both tasks (ML12 and ML13) for all four entities (*Disease*, *Gene*, *SNP*, *Chemical*). Each task was about to be performed for different batch ranges and ten times per batch. More specifically, seven different batch ranges were chosen. Batches of 100, 500, 1,000, 1,500, 2,000, 2,500 and 3,000 sentences. The reason for this choice had to do with the evaluation of the change of the *Accuracy*, *Precision* and *Recall* scores as the number of sentences was increasing.

For each batch, the code would have to be run ten times for ten different batches of the same size, in order to extract the average value and thus a more representative result for each case.

However, as it can be seen from table 1, the size of the dataset that represents the number of sentences that have at least one *SNP* entity in general is 3,432. This means that tasks ML12 and ML13 cannot be performed for the *SNPs*, due to the fact that dataset size is not big enough, so as to end up to reliable results/conclusions. Thus, the two tasks were performed for the rest three entities (*Disease*, *Gene*, *Chemical*).

It is important to be mentioned that the reason why the maximum batch size that was chosen is 3,000 sentences has to do with the computational power of the system. Google Colab provided a 12 GB RAM, powerful enough to run the code for no more than 3,000 sentences. Of note that the size of each sentence plays a key role as well to the number of the permitted input sentences. The longer the sentences, the smaller the maximum input batch size. As already mentioned in Chapter 3, only sentences that were comprised of more than 15 and less than 300 letters were extracted. It was observed that less than 15 did not constitute representative sentences. Looking at the datasets, we find that they have many sentences like ‘0.26 * 0.24 *’, ‘(n=4)’ or sentences that correspond to titles

of paragraphs, or the declaration of the papers' authors. These sentences have no practical meaning and introduce noise that interferes with the model to extract features. So, the dataset was reduced by excluding those "sentences" and the noise is also reduced, thus improving the performance of the models.

As far as the maximum size of sentences is concerned, it has to be noted that sentences with more than 300 letters could be included. However, this would have as a result to run the Machine Learning process with less than 3,000 sentences, due to the confined computational power of the system. Sentences of up to 200 letters would allow the system to take as input datasets of about 4,000 sentences, but on the other side, the quality of these sentences wouldn't be the best possible, leading to confusing and uncertain results after the BERT model process. All in all, a happy medium was found between choosing the right size of sentences and the right dataset size.

Below, tables 2, 3 and 4 depict the average *Accuracy*, *Precision*, *Recall*, as well as the average execution time for each batch size, of each of the two tasks, of each of the three entities.

The acquired data presented below were plotted in order to get a visual depiction of the results, making it easier to interpret it and reach to conclusions (see Fig. 48, 49, 50, 51, 52, 53).

Table 2: Evaluation scores for the *Disease* entity.

Disease					
ML12		Accuracy	Precision	Recall	Duration
Number of sentences	100	0.748	0.79	0.77	25''
	500	0.759	0.81	0.78	1'05''
	1,000	0.778	0.823	0.79	1'30''
	1,500	0.793	0.841	0.814	3'
	2,000	0.8	0.84	0.823	3'50''
	2,500	0.81	0.86	0.836	4'30''
	3,000	0.829	0.89	0.846	5'
ML13		Accuracy	Precision	Recall	Duration
Number of sentences	100	0.856	0.841	0.882	30''
	500	0.899	0.903	0.905	1'20''
	1,000	0.9	0.901	0.903	2'
	1,500	0.907	0.907	0.906	3'35''
	2,000	0.911	0.913	0.909	4'30''
	2,500	0.915	0.914	0.917	6'
	3,000	0.909	0.91	0.909	7'

Table 3: Evaluation scores for the Gene entity.

Gene					
ML12		Accuracy	Precision	Recall	Duration
Number of sentences	100	0.668	0.651	0.666	35''
	500	0.789	0.763	0.791	1'25''
	1,000	0.819	0.818	0.813	3'
	1,500	0.832	0.826	0.838	6'
	2,000	0.828	0.823	0.827	4'20''
	2,500	0.834	0.836	0.834	4'50''
	3,000	0.847	0.838	0.845	6'
ML13		Accuracy	Precision	Recall	Duration
Number of sentences	100	0.768	0.735	0.752	15''
	500	0.84	0.835	0.826	45''
	1,000	0.839	0.822	0.84	1'40''
	1,500	0.86	0.853	0.854	2'30''
	2,000	0.866	0.847	0.858	3'30''
	2,500	0.871	0.873	0.861	3'55''
	3,000	0.872	0.862	0.858	5'

Table 4: Evaluation scores for the Chemical entity.

Chemical					
ML12		Accuracy	Precision	Recall	Duration
Number of sentences	100	0.54	0.579	0.551	14''
	500	0.608	0.612	0.635	45''
	1,000	0.65	0.669	0.654	1'40''
	1,500	0.668	0.681	0.673	2'20''
	2,000	0.687	0.714	0.681	3'
	2,500	0.705	0.729	0.7	4'20''
	3,000	0.701	0.725	0.702	4'40''
ML13		Accuracy	Precision	Recall	Duration
Number of sentences	100	0.796	0.819	0.778	12''
	500	0.87	0.868	0.876	45''
	1,000	0.892	0.886	0.889	1'40''
	1,500	0.898	0.89	0.907	2'50''
	2,000	0.902	0.893	0.904	3'55''
	2,500	0.908	0.908	0.91	4'55''
	3,000	0.91	0.907	0.914	5'35''

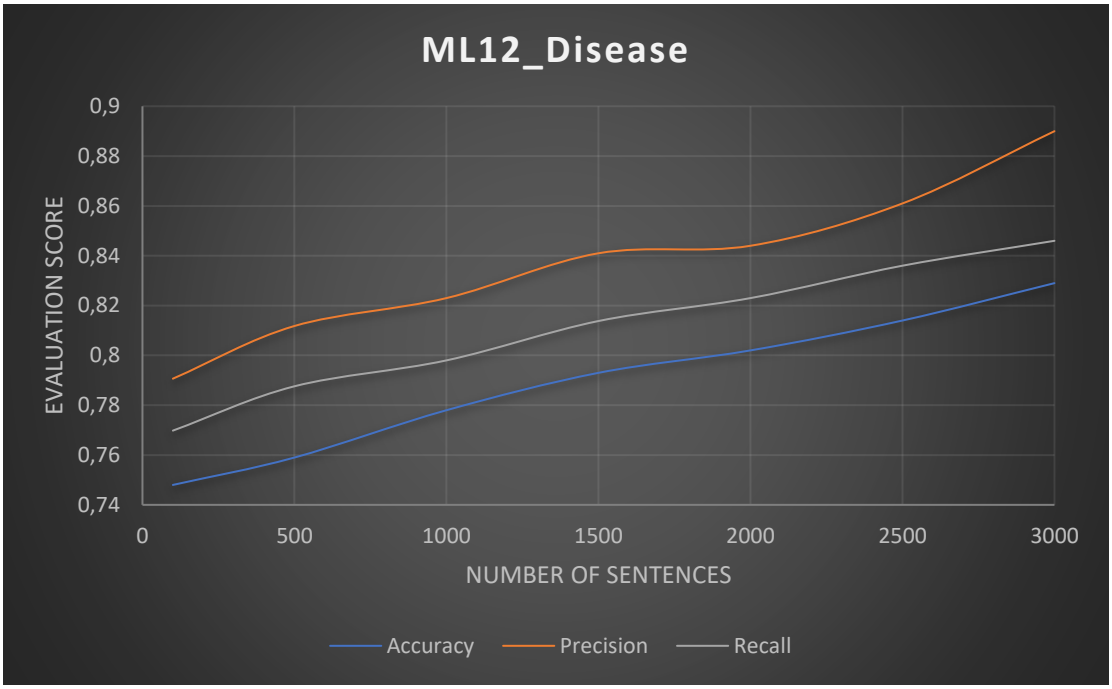


Fig. 48: Evaluation scores of the three metrics (*Accuracy*, *Precision*, *Recall*) for different numbers of sentences (entity: *Disease*, task: *ML12*).

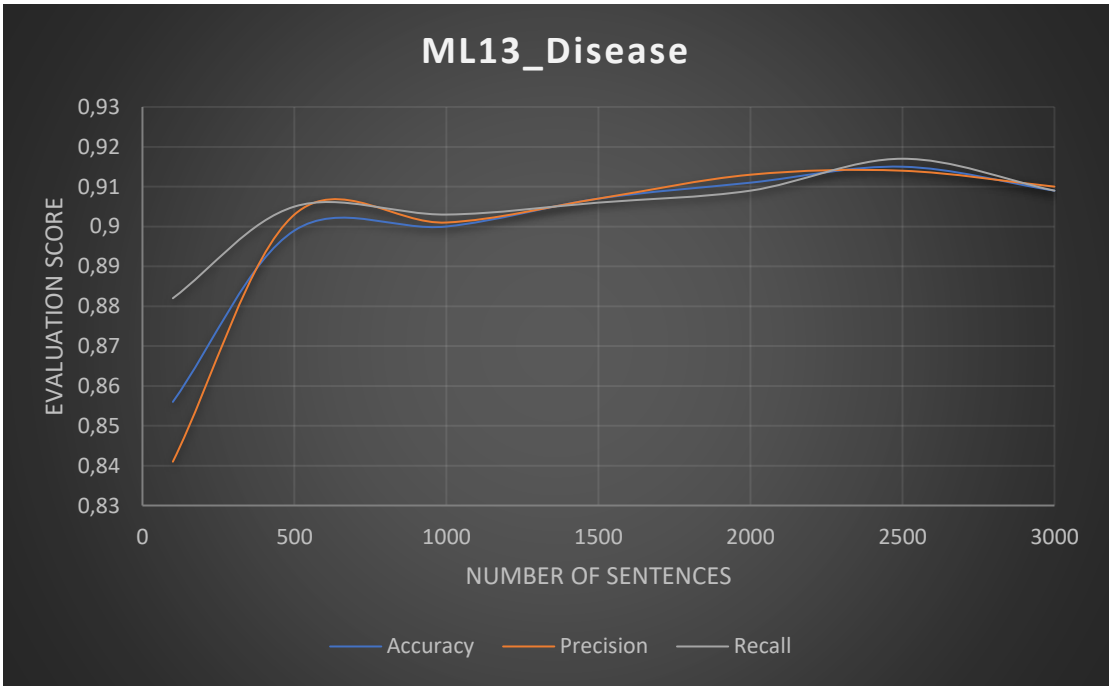


Fig. 49: Evaluation scores of the three metrics (*Accuracy*, *Precision*, *Recall*) for different numbers of sentences (entity: *Disease*, task: *ML13*).

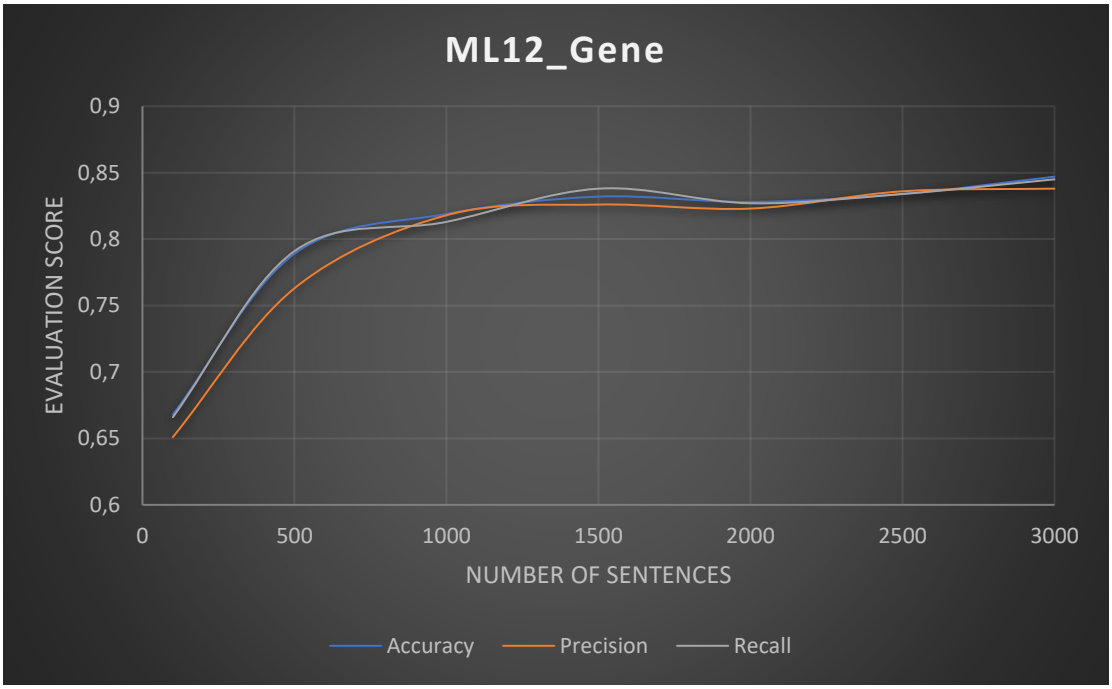


Fig. 50: Evaluation scores of the three metrics (*Accuracy*, *Precision*, *Recall*) for different numbers of sentences (entity: *Gene*, task: *ML12*).

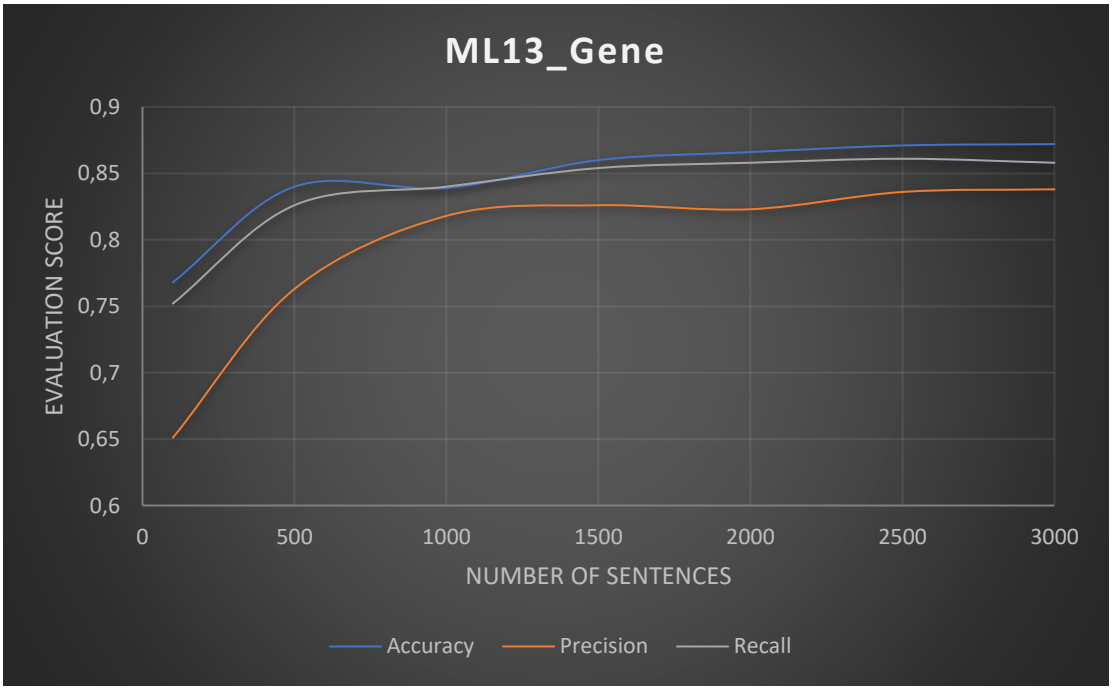


Fig. 51: Evaluation scores of the three metrics (*Accuracy*, *Precision*, *Recall*) for different numbers of sentences (entity: *Gene*, task: *ML13*).

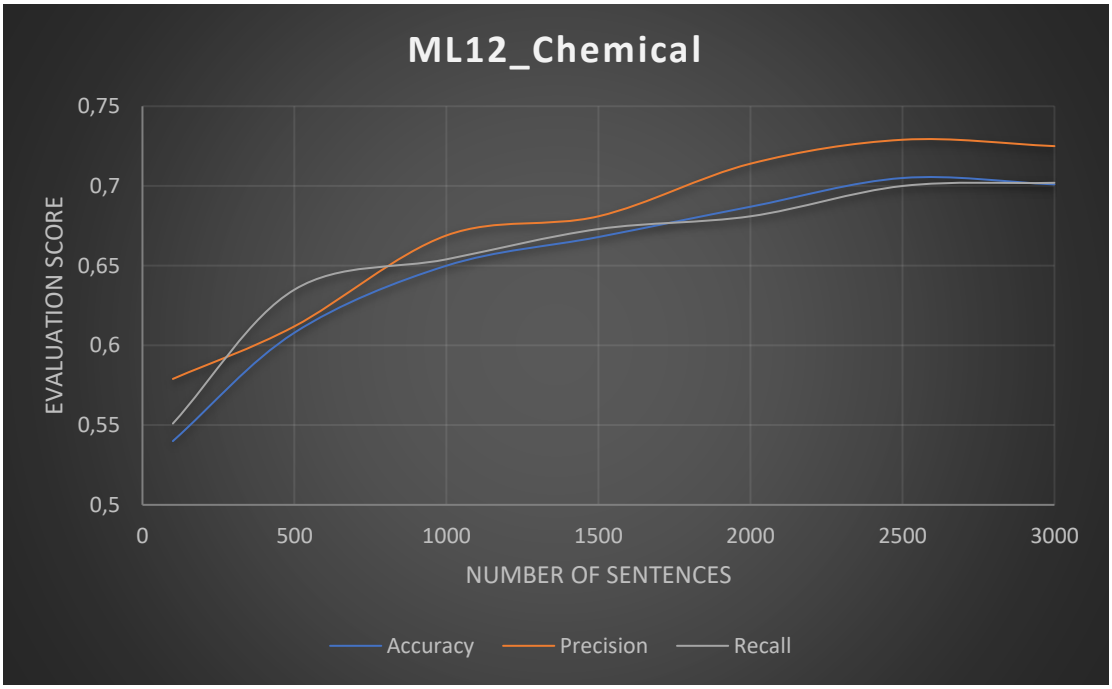


Fig. 52: Evaluation scores of the three metrics (*Accuracy*, *Precision*, *Recall*) for different numbers of sentences (entity: *Chemical*, task: *ML12*).

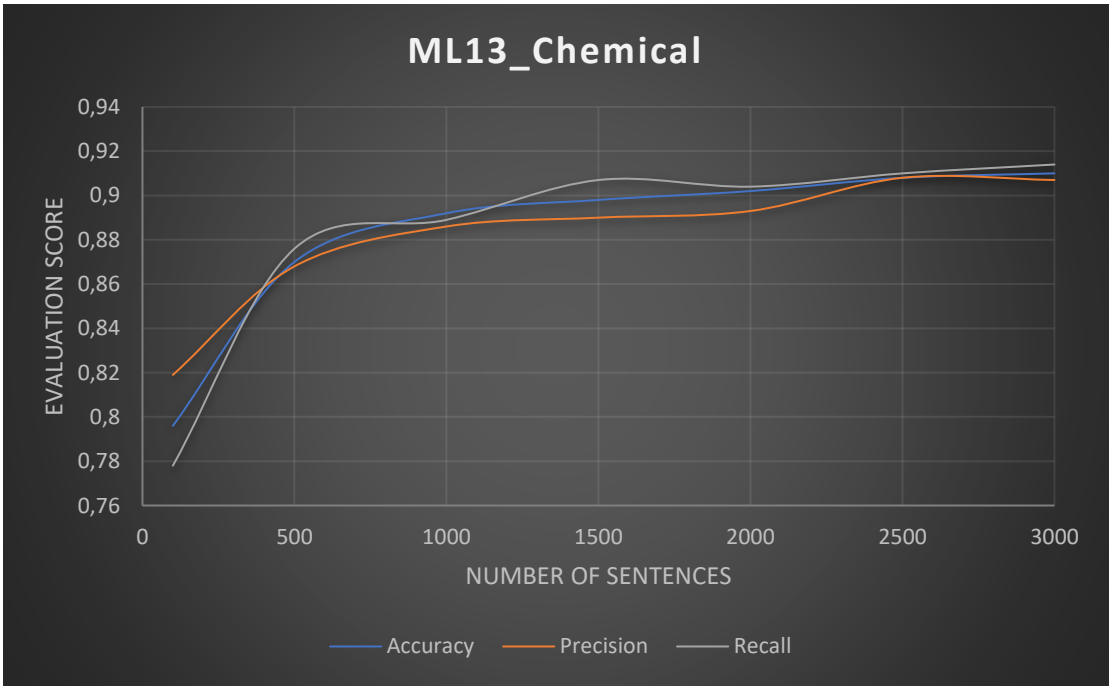


Fig. 53: Evaluation scores of the three metrics (*Accuracy*, *Precision*, *Recall*) for different numbers of sentences (entity: *Chemical*, task: *ML13*).

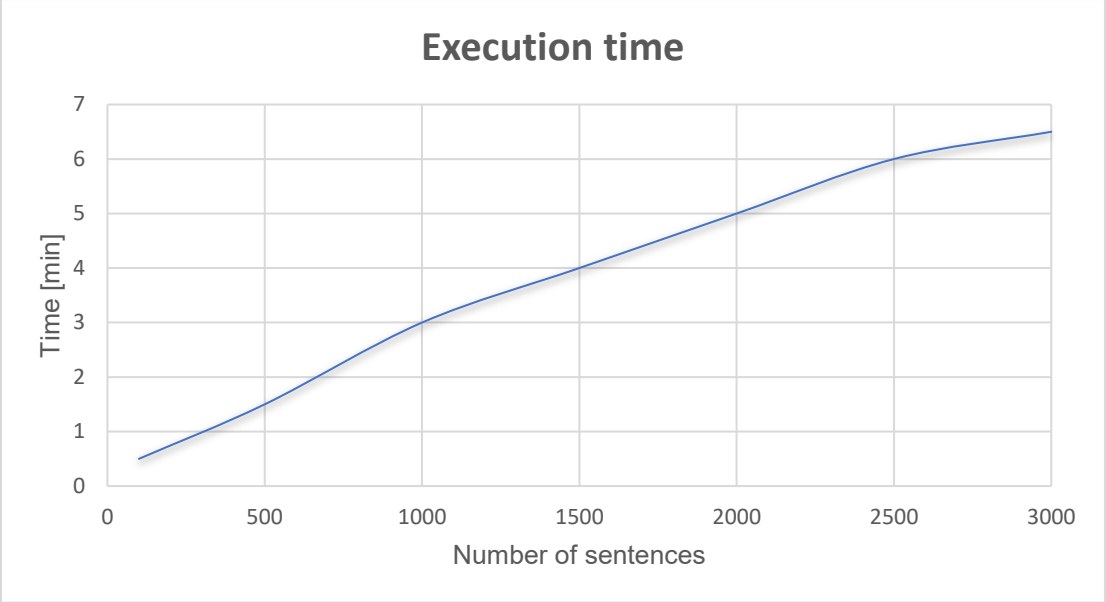


Fig. 54: Execution time versus different batch sizes.

Fig. 54 depicts the average time – among ten different executions of the model per n sentences, where n = 100, 500, 1,000, 1,500, 2,000, 2,500, 3,000 – needed to run each model for different dataset sizes. Of course, the more the sentences the longer the execution time, whose curve seems to present a linear behavior.

If, instead of distilBERT, BERT was applied, the increase in execution time would be so big that the percentage of its score increase would be negligible for the purpose of this study.

Thanks to Google Colab and its free providence of such a powerful GPU, less time is spent waiting for the execution of the code, and the computational power is very big, paving the road for more systematic research.

Chapter 5: Discussion and analysis of findings

Many useful remarks can be made by observing the plots of the previous Chapter. By comparing the two tasks, ML12 and ML13, for all entities, it can be observed that ML13 task achieves higher scores. On average, ML13 tasks' scores are ~5-7% higher, especially for the larger datasets. Moreover, ML13 tasks' curves present an early flattening already from datasets with 500 or 1,000 sentences. More specifically, the *ML13_Disease* task reaches ~90% score already from 500 sentences dataset and *ML13_Gene* and *ML13_Chemical* tasks reach ~86% and ~90% scores, respectively, for dataset sizes of 1,000 sentences. This means that no more sentences are needed for this task to reach its maximum potential.

The model performs better when the ML13 dataset is given as input because it has to retrieve a specific biomedical entity between sentences that include one entity and sentences that have no entities at all. The ML12 task makes things worse, because the second category of labeled sentences also includes entities and more specifically, the other two entities and not the one we are searching for. This makes it more difficult for the model to decipher all these entities.

As far as the ML12 task is concerned, when it comes to the *Gene* entity, it can be noticed that the curve flattens after 1,000 sentences, so no larger dataset is needed. However, when it comes to the *ML12_Chemical* task, the evaluation scores seem to start stabilizing after datasets of 2,500 sentences. Nevertheless, larger batches of 3,500 or 4,000 sentences could help assert this conclusion. Finally, the model that behaves differently compared to the other tasks is the *ML12_Disease* task, where we see that, as the dataset gets larger, the scores keep increasing as well. That means that our larger dataset of 3,000 sentences is not sufficient and thus, larger batches are needed in order to reach to trustworthy evaluations and of course, more efficiently classify sentences that include *Disease* entities.

Finally, one last noteworthy observation has to do with the fact that the *ML12_Disease* task scores are ~10-20% lower than the rest tasks' scores, meaning that the model "struggles" more to classify sentences that contain a *Disease* entity and sentences that contain all other entities but D*isease* ones.

Chapter 6: Challenges and future work

These days, more and more datasets are created which contain the same bioentity types, but usually different annotations. For instance, a dataset might label an entity as a *Protein* entity, whereas another one might consider it to be a *Gene* entity. Another issue has to do with the fact that some biomedical formats consider a long entity of many words as one entity, whereas other datasets split this entity to two or more entities. These two problems deteriorate the accuracy of the models and that is why a uniform annotation standard needs to be developed.

BioNER requires a large amount of annotated training data, many samples and of high quality. Many high-quality and high-quantity corpora for bioentities already exist. However, there is still a lack of corpora that contain entities, such as *Mutations*, *Species* and of course, as we noticed above, *SNPs*. The larger the dataset size, the better the recognition of these entities and consequently, the better the performance of the model.

Thus, the pre-processing part of the code that was analyzed in Chapter 3 could also be applied for new biomedical entities, such as *Mutations*, in order to enrich the number of different biomedical datasets. Of course, after the pre-processing, these datasets would pass through the BERT model, as was the case for the three entities of this study.

As far as the *SNPs* are concerned, as already mentioned, although 1,000 BioC files were parsed, which correspond to 100,000 papers, the sentences that contained *SNP* entities were not so many, in order to create datasets big enough to extract safe conclusions. So, as a future work, much more files could be parsed, maybe 2,000-3,000, which correspond to about 200,000-300,000 papers, in order to extract more sentences and perform these classification tasks for this type of entity, too.

As it was stated in Chapter 4, for each task and entity the model was run for different batch sizes and ten times per batch size. For each iteration, *Accuracy*, *Precision* and *Recall* were calculated and in the end of the ten iterations, an average value of these metrics was taken. However, in some cases, especially for small size datasets, the disparity of these values among each iteration was very big. Thus, more iterations would stabilize the moving average of these values.

Another future work that would improve the model's ability to classify these sentences and distinguish among the bioentities is the addition of new kind of datasets. One idea would be to keep the sentences that create only one type of entity and then compare them among each other. Thus, 3 new datasets could be created. One dataset with sentences that contain only *Chemicals* versus sentences that contain only *Genes*. Another one with sentences that contain only *Genes* versus sentences that contain

only *Diseases* and the last one with *Diseases* versus *Chemicals*. Those three new tasks would contribute to better distinguish classes of bioentities in biomedical text.

By directly applying the state-of-the-art NLP methods to biomedical sources could lead to poor results, due to a word distribution drift from the general domain corpora to corpora of the biomedical domain, ending up to linguistic ambiguities. To overcome these challenges and generally improve this study's model, bioBERT, the domain specific deep neural network model could be used. As we know from section 2.2.5, bioBERT is pre-trained on biomedical domain corpora in order to handle the linguistic challenges within biomedical literature. According to [30], bioBERT obtains higher scores in biomedical NER than the current state-of-the-art models. BioBERT can recognize biomedical named entities that BERT cannot and can find the exact boundaries of named entities. This gave rise to enhanced performance on many biomedical NLP tasks [40].

Moreover, since in the scope of this study, the experiments were performed using the distilBERT model, an increase of the model size would be expected to further improve the results. Therefore, a future work would be to re-run the experiments using BERT-Large, which requires even more computation time.

The only change in the code would be the following:

```
model_class, tokenizer_class, pretrained_weights =
(ppb.DistilBertModel,  ppb.DistilBertTokenizer, 'distilbert-base-uncased')
```

↓

```
model_class, tokenizer_class, pretrained_weights =
(ppb.BertModel, ppb.BertTokenizer, 'bert-base-uncased')
```

The highest accuracy score for these datasets is between 96-97%. DistilBERT can be trained to improve its score on this task, thanks to a process called fine-tuning. Through fine-tuning, BERT's weights are updated in order to improve its performance in the sentence classification.

One more way to improve the model is by setting different parameters that are given as input to the *LogisticRegression* function, which represents the classification model [see Chapter 3 and 4]. For example, by setting the regularization strength C equal to 10.0, instead of the default value of 1.0, another model is created with different parameters and consequently, a different probability matrix and a different set of coefficients and predictions. The absolute values of the intercept b_0 and the coefficient b_1 become larger, because of the larger C value, which means weaker regularization, or weaker penalization related to high b_0 and b_1 values. The change of the b_0 and b_1 values, has as a result

a change of the linear function f(x), different probabilities p(x) and a different regression line. Also, the classification performance and other predicted outputs might change as well. The confusion matrix changes and thus the *Accuracy*, the *Recall* and the *Precision* metrics get higher, provided that the new parameters are tuned correctly.

A very common challenge is to exceed the already fairly large amount of text that fits into the BERT input. Unfortunately, there is no obvious solution to this problem. However, there are a number of different ideas that could be tried. According to [41], dropping some of the tokens will not affect the performance of the task. The tokens could be dropped from the beginning of the text, the end of it or keep some of the beginning, some of the end and cut out some of the middle. These three truncation methods are described as *head-only*, meaning that the first 510 tokens of the text are kept. This number of tokens means that we have to leave space for the special [CLS] token in the beginning and the [SEP] token in the end. Keeping just the last tokens is *tail-only*. *Head+tail* is cutting some of the middle tokens. Some different numbers were tried and it was found out that picking the first 128 tokens plus the last 382 tokens, worked the best.

Another strategy we could take is to break the text into multiple chunks, process the chunks separately and then combine the results. Let's say we have a piece of text that is 1,200 tokens long and we break it into two chunks of 512 tokens each and then one more of 176 for the balance. So, in the same paper, [41], the text is divided in chunks and for each of those chunks it is sent through the BERT model. An embedding is extracted for the chunk and then the embeddings are pooled together. Minimum pooling, maximum pooling and averaging were tried.

One more idea would be text summarization, meaning the condensation of the document into a smaller length that BERT can handle. There are two types of text summarization: extractive and abstractive. Extractive means that you go through the document and pick out the most pertinent sentences. Then you return those as the summarization. So, you don't modify the text. You just kind of pick out the most pertinent text. Also, you typically get to pick the number of sentences that you want to summarize the document down to, thus you have the ability to keep a lot of text in the summarization strategy. Abstractive means that you are allowed to generate new text to try and summarize.

Chapter 7: Conclusion

The aim of this study was to recognize four bioentities (*Disease*, *Gene*, *Chemical, SNP*) in biomedical text using the state-of-the-art BERT technique/model – for classification of sentences.

The datasets used were created by parsing and processing BioC XML files.

The results showed that an appropriately pre-trained BERT model delivers state-of-the-art recognition performance in many cases, without extensive fine-tuning and optimization requirements, outperforming previous models on the NER biomedical text mining task.

This outcome encourages further tuning, new methodologies and generates new challenges. Therefore, the experiments could be re-run dealing with issues stated in the previous Chapter and applying the “future work” goals, also stated above.

References

1. IBM Cloud Education (2020, July 2), "Natural Language Processing (NLP)" [Blog post], Available at: https://www.ibm.com/cloud/learn/natural-language-processing?utm_medium=OSocial&utm_source=Youtube&utm_content=000027BD&utm_term=10004432&utm_id=YTDescription-101-What-is-NLP-LH-Natural-Language-Processing-Guide (Accessed: 2021, December 3).
2. Google AI Blog – The latest from Google Research (2018, November), "Open Sourcing BERT: State-of-the-Art Pre-training for Natural Language Processing" [Blog post], Available at: https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html (Accessed: 2021, November 22).
3. HuggingFace (2019, August), "Smaller, faster, cheaper, lighter: Introducing DistilBERT, a distilled version of BERT" [Blog post], Available at: https://medium.com/huggingface/distilbert-8cf3380435b5 (Accessed: 2021, November 22).
4. Sanh V., Debut L., Chaumond J., Wolf T. (2020, March), "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter", arXiv:1910.01108.
5. "Natural language processing" (2021, November 21), Wikipedia, Available at: https://en.wikipedia.org/wiki/Natural_language_processing, (Accessed: 2021, November 20).
6. Simplilearn (2021, March 17), Natural Language Processing In 5 Minutes | What Is NLP And How Does It Work? [Video File], Available at: https://www.youtube.com/watch?v=CMrHM8a3hqw&ab_channel=Simplilearn (Accessed: 2021, November 25).
7. "Biomedical text mining" (2021, August 22), Wikipedia, Available at: https://en.wikipedia.org/wiki/Biomedical_text_mining, (Accessed: 2021, November 29).
8. Friedman C., Anderson P. O., Austin J. H. M., Cimino J. J., Johnson S. B. (1994, April), "A general Natural-language Text Processor for Clinical Radiology", Journal of the American Medical Informatics Association, volume 1 [2].
9. Comeau D. C., Wei C. - H., Doğan R. I., Lu Z. (2019, September), "PMC text mining subset in BioC: about three million full-text articles and growing", Bioinformatics, volume 35 [18], pp. 3533–3535.
10. Zhang S., Elhadad N. (2013, December), "Unsupervised biomedical named entity recognition: Experiments with clinical and biological texts", Journal of Biomedical Informatics, volume 46 [6], pp. 1088-1098.

11. Wei C. - H., Allot A., Leaman R., Lu Z. (2019, May), "PubTator central: automated concept annotation for biomedical full text articles", Nucleic Acids Research, volume 47 [1], pp. 587-593, doi: https://doi.org/10.1093/nar/gkz389.
12. Wei C. – H., Kao H. – Y., Lu Z. (2013, July), "PubTator: a web-based text mining tool for assisting biocuration", Nucleic Acids Research, volume 41, pp. 518-522, doi: 10.1093/nar/gkt441.
13. Comeau D. C., Doğan R. I., Ciccarese P., Cohen K. B., Krallinger M., Leitner F., Lu Z, Peng Y., Rinaldi F., Torii M., Valencia A., Verspoor K., Wiegers T. C., Wu C. H., Wilbur W. J. (2013, September), "BioC: a minimalist approach to interoperability for biomedical text processing", Database: The journal of biological databases and curation, volume 2013, doi: https://doi.org/10.1093/database/bat064.
14. Serrano.Academy (2016, September 9), A Friendly Introduction to Machine Learning [Video File], Available at: https://www.youtube.com/watch?v=IpGxLWOIZy4&ab_channel=Serrano.Academy (Accessed: 2021, November 25).
15. Simplilearn (2018, February 13), Machine Learning Tutorial | Machine Learning Basics | Machine Learning Algorithms [Video File], Available at: https://www.youtube.com/watch?v=G7fPB4OHkys&ab_channel=Simplilearn (Accessed: 2021, November 22).
16. MIT OpenCourseWare (2017, May 19), 11. Introduction to Machine Learning [Video File], Available at: https://www.youtube.com/watch?v=h0e2HAPTGF4&ab_channel=MITOpenCourseWare (Accessed: 2021, November 29).
17. Google Cloud Tech (2017, August 25), What is Machine Learning? [Video File], Available at: https://www.youtube.com/watch?v=HcqpanDadyQ&ab_channel=GoogleCloudTech (Accessed: 2021, November 29).
18. KD Nuggets™ (2018, May), "Frameworks for Approaching the Machine Learning Process" [Blog post], Available at: https://www.kdnuggets.com/2018/05/general-approaches-machine-learning-process.html (Accessed: 2021, December 3).
19. Michael Nielsen (2019, December), "Neural Networks and Deep Learning" [Blog post], Available at: http://neuralnetworksanddeeplearning.com/index.html (Accessed: 2021, December 1).
20. freeCodeCamp.org (2019, April 16), How Deep Neural Networks Work - Full Course for Beginners [Video File], Available at: https://www.youtube.com/watch?v=dPWYUELwIdM&ab_channel=freeCodeCamp.org (Accessed: 2021, November 29).
21. Yuxi Li (2018, November), "Deep Reinforcement Learning: An Overview", arXiv:1701.07274.

22. Cybenko G. (1989, December), “Approximation by superpositions of a sigmoidal function”, Mathematics of Control, Signals and Systems, volume 2, pp. 303-314.
23. Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L., Polosukhin I. (2017), “Attention is all you need”, 31st Conference on Neural Information Processing Systems, Long Beach, CA, USA.
24. “Word embedding” (2021, November 19), Wikipedia, Available at: https://en.wikipedia.org/wiki/Word_embedding, (Accessed: 2021, November 20).
25. Hochreiter S., Schmidhuber J. (1997), “Long Short-Term Memory”, Neural Comput, volume 9 [8], pp. 1735–1780, doi: https://doi.org/10.1162/neco.1997.9.8.1735.
26. Google – The Keyword (2019, October), “Understanding searches better than ever before” [Blog post], Available at: https://www.blog.google/products/search/search-language-understanding-bert/ (Accessed: 2021, November 22).
27. Devlin J., Chang M. – W., Lee K., Toutanova K. (2019, May), “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, arXiv:1810.04805.
28. Peters M. E., Neumann M., Iyyer M., Gardner M., Clark C., Lee K., Zettlemoyer L. (2018, June), “Deep contextualized word representations”, Proceedings of NAACL-HLT 2018, New Orleans, Louisiana, Association for Computational Linguistics, pp. 2227–2237.
29. Jay Alammar (2018, December), “Visualizing machine learning one concept at a time” [Blog post], Available at: https://jalammar.github.io/illustrated-bert/ (Accessed: 2021, November 25).
30. Lee J., Yoon W., Kim S., Kim D., Kim S., So C. H., Kang J. (2019, September), “BioBERT: a pre-trained biomedical language representation model for biomedical text mining”, Bioinformatics, volume 36 [4], pp. 1234-1240, doi: 10.1093/bioinformatics/btz682.
31. Rajpurkar P., Zhang J., Lopyrev K., Liang P. (2016), “Squad: 100,000þ questions for machine comprehension of text”, Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, Austin, TX., pp. 2383–2392, Association for Computational Linguistics, https://www.aclweb.org/anthology/D16-1264.
32. Wiese G., Weissenborn D., Neves M. (2017, June), “Neural domain adaptation for biomedical question answering”, Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017), Vancouver, Canada, pp. 281–289, Association for Computational Linguistics. https://www.aclweb.org/anthology/K17-1029.
33. Google Colab, “What is Colaboratory?” [Blog post], Available at: https://colab.research.google.com/ (Accessed: 2021, November 15).
34. Kluyver T., Ragan-Kelley B., Perez F., Granger B., Bussonnier M., Frederic J., et al. (2016), “Jupyter Notebooks – a publishing format for reproducible computational workflows”, Loizides F, Schmidt

B, editors, Positioning and Power in Academic Publishing: Players, Agents and Agendas, p. 87–90, doi: 10.3233/978-1-61499-649-1-87.

35. “scikit-learn” (2021, November 20), Wikipedia, Available at: https://en.wikipedia.org/wiki/Scikit-learn, (Accessed: 2021, November 10).
36. RealPython (2019), “Logistic Regression in Python” [Blog post], Available at: https://realpython.com/logistic-regression-python/ (Accessed: 2021, December 2).
37. Pedregosa F., Varoquaux G., Gramfort A., Michel V., Thirion B., Grisel O., Blondel M., et al. (2011), “Scikit-learn: Machine Learning in Python”, Journal of Machine Learning Research, volume 12, pp. 2825-2830.
38. Analytics Vidhya (2020, April), “Everything you Should Know about Confusion Matrix for Machine Learning” [Blog post], Available at: https://www.analyticsvidhya.com/blog/2020/04/confusion-matrix-machine-learning/ (Accessed: 2021, November 29).
39. Song B., Li F., Liu Y., Zeng X. (2021), “Deep learning methods for biomedical named entity recognition: a survey and qualitative comparison”, Briefings in Bioinformatics, 00 [0], pp. 1–18, doi: https://doi.org/10.1093/bib/bbab282.
40. Naseem U., Musial K., Eklund P. W., Prasad M. (2020, August), “Biomedical Named-Entity Recognition by Hierarchically Fusing BioBERT Representations and Deep Contextual-Level Word-Embedding”, doi: 10.1109/IJCNN48605.2020.9206808.
41. Sun C., Qiu X., Xu Y., Huang Y. (2020), “How to Fine-Tune BERT for Text Classification? arXiv:1905.05583.