UNIVERSITY OF PIRAEUS

School of Information and Communication Technologies

Department of Informatics

Thesis

Thesis Title: Τίτλος Διατριβής:	**Natural Language Processing and Text Classification with Bert & BiLSTM model** Επεξεργασία Φυσικής Γλώσσας και Ταξινόμηση Κειμένων με Μοντέλο Bert & BiLSTM
Student's name-surname:	**Giannis Touloupis**
Father's name:	**Andreas**
Student's ID No:	**P18217**
Supervisor:	**Dionysios Sotiropoulos, Professor**

September 2024/ Σεπτέμβριος 2024

Copyright ©

Summary

This thesis refers to the architecture of deep learning models . This is accomplished by using the deep learning models to classify tweets related to Covid-19 to three different categories. The first category as tweets which are in favor of vaccination, agree and implement methods of protection against the virus. The second category as tweets which are against the vaccination and promote conspiracy theories. The third category as tweets which they have a neutral stance.

Specifically, to solve the problem above they have been collected a number of tweets so we can train the two deep learning models we are using, a Bidirectional LSTM and a Bert model. These models are going to be analyzed and explained through every step and at the same time we will emphasize some differences between those two models.

To train these models we have firstly to collect those tweets and categorize them, also during this step we had to find another collection method due to twitter policy changes. In addition, we will present the preparation of those tweets so that the models can accept them and use them to train as best as possible.

The whole preparation and the process is described in detail later on and the results and any observations of the models at the end.

Key words: AI ,machine learning model ,BiLSTM, Bert, transformers, tweets, dataset, covid, vaccine, Optimizers, data, training

Περίληψη

Αυτή η διπλωματική εργασία αναφέρεται στην αρχιτεκτονική των μοντέλων βαθιάς μάθησης. Αυτό επιτυγχάνεται με τη χρήση μοντέλων βαθιάς μάθησης για την ταξινόμηση των tweets που σχετίζονται με τον Covid-19 σε τρεις διαφορετικές κατηγορίες. Η πρώτη κατηγορία αφορά tweets που είναι υπέρ του εμβολιασμού, συμφωνούν και εφαρμόζουν μεθόδους προστασίας κατά του ιού. Η δεύτερη κατηγορία αφορά tweets που είναι κατά του εμβολιασμού και προωθούν θεωρίες συνωμοσίας. Η τρίτη κατηγορία αφορά tweets που έχουν ουδέτερη στάση.

Συγκεκριμένα, για την επίλυση του παραπάνω προβλήματος, συλλέχθηκε ένας αριθμός tweets ώστε να εκπαιδεύσουμε τα δύο μοντέλα βαθιάς μάθησης που χρησιμοποιούμε, ένα Δικτυακό Μοντέλο Bidirectional LSTM και ένα μοντέλο Bert. Αυτά τα μοντέλα θα αναλυθούν και θα εξηγηθούν βήμα-βήμα, και ταυτόχρονα θα δοθεί έμφαση σε κάποιες διαφορές μεταξύ τους.

Για να εκπαιδεύσουμε αυτά τα μοντέλα, έπρεπε πρώτα να συλλέξουμε και να κατηγοριοποιήσουμε τα tweets. Κατά τη διάρκεια αυτού του βήματος χρειάστηκε να βρούμε μια νέα μέθοδο συλλογής λόγω αλλαγών στην πολιτική του Twitter. Επιπλέον, θα παρουσιάσουμε την προετοιμασία αυτών των tweets ώστε τα μοντέλα να μπορούν να τα δεχθούν και να τα χρησιμοποιήσουν για εκπαίδευση με τον καλύτερο δυνατό τρόπο.

Η όλη προετοιμασία και η διαδικασία περιγράφονται λεπτομερώς παρακάτω, ενώ στο τέλος παρουσιάζονται τα αποτελέσματα και οι παρατηρήσεις σχετικά με τα μοντέλα.

Λέξεις-κλειδιά: AI , machine learning model,BiLSTM, Bert, transformers, tweets, dataset, covid, vaccine, Optimizers, data, training, model

Acknowledgement

I would like to express my deepest gratitude advisor to my supervising professor Mr. Dionysios Sotiropoulos, for his unwavering support, insightful guidance, continuous encouragement and most importantly his patience and time that he invested. His expertise and constructive feedback have been instrumental in shaping the direction and quality of this research.

Table of Contents

Chapter 1 8
1.1 Introduction to artificial intelligence 8
1.2 Introduction and description of the problem 12
Chapter 2 14
2.1 Bert model 15
2.2 Bidirectional LSTM 17
2.3 Natural Language Processing 18
2.4 Text translation with BiLSTM and Bert model and their differences 19
2.5 Text Summarization with BiLSTM and Bert model and their differences 20
2.6 Sentiment Analysis with BiLSTM and Bert model and their differences 21
2.7 Text Classification 22
Chapter 3 25
3.1 The importance of a dataset 25
3.2 Dataset collection 26
3.3 Dataset preparation 28
3.4 Graphs and Charts 30
3.5 Dataset preparation for Bert Model 35
3.6 Dataset preparation for BiLSTM Model 38
Chapter 4 41
4.1 Bert model creation 41

4.2 Bert model initializations 42
4.2 Bert model training 45
4.3 Bert model testing evaluation 53
Chapter 5 56
5.1 BiLSTM model creation. 56
5.2 BiLSTM model initializations. 57
5.3 AdaDelta Optimizer. 58
5.4 The key differences between AdaDelta and AdamW. 60
5.5 BiLstm architecture. 62
5.6 BiLstm training. 67
Chapter 6 70
6.1 Conclusions. 70
6.2 Improvements and Future work. 70
6.3 Bibliography. 71

Table of Images

1. Supervised learning schema 9
2. Unsupervised learning schema 10
3. Reinforcement learning schema. 11
4. Machine learning vs Deep learning schema 12
5. Bert Model Diagram 16
6. Bidirectional LSTM model diagram 17
7. Python script to extract tweets based on hashtags. 27
8. Preparation steps 27
9. Pandas import and dataset load 28
10. Dataset format 29
11. Cleaning data and checking for null values 30
12. Creating graphs and charts 31
13. Word Frequency for Pro-Vaxx Tweets 32
14. Word Frequency for Anti-Vaxx Tweets 32
15. Word Frequency for Neutral Tweets 33

16. Word Frequency for Pro-Vaxx Tweets with custom chart 33
17. Word Frequency for Anti-Vaxx Tweets with custom chart 34
18. Word Frequency for Neutral Tweets with custom chart 34
19. Dataset preparation for the Bert model 1.1 35
20. Dataset preparation for the Bert model.1.2 37
21. Dataset preparation for the BiLSTM model, ... 40
22. Installation of transformers, ... 42
23. Function importation for Bert model 43
24. Initializations for the Bert model. 44
25. Building of the Bert model 1.1 46
26. Bert's training early stopping 52
27. Bert's training 53
28. Bert's results 55
29. BiLstm Libraries 58
30. BiLstm Architecture 63
31. BiLstm Training 67
32. BiLstm Training Results 68
33. Creation of dummy data 68
34. Prediction results on the dummy data 69

Chapter 1

1.1 Introduction to artificial intelligence

Our brain from the first stages of its development at a very young age begins to learn about the results something has on the environment and the environment on us. For example, even what we see with our optical system is a prediction of what we are looking to and it is not always precise. The precision of our predictions is increasing by having more examples in our life from stimuli we receive.

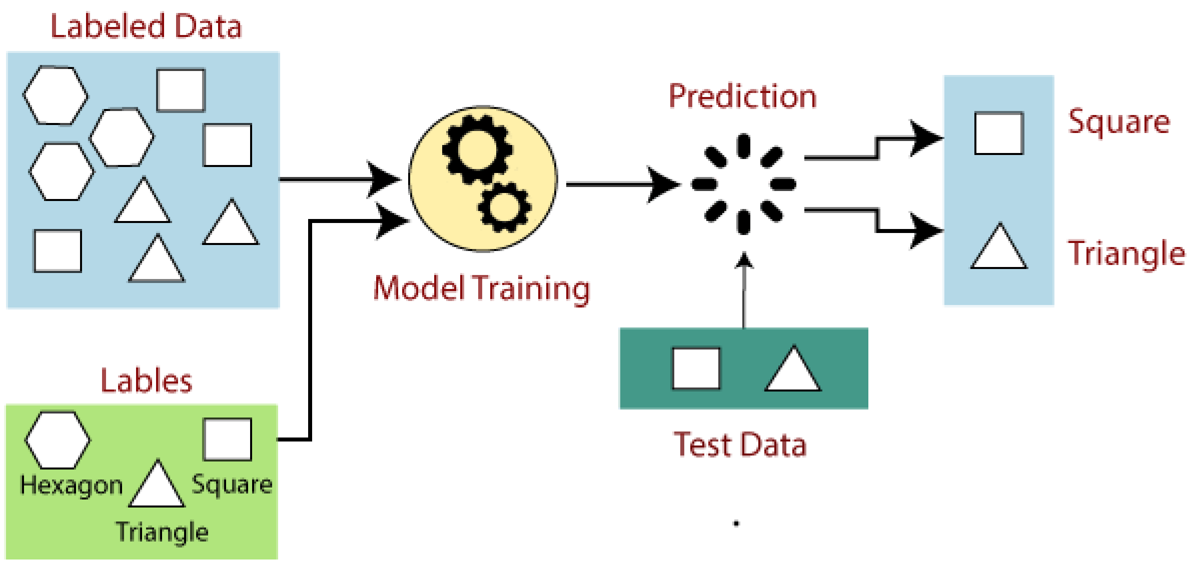
Neural networks work the same way, by receiving a set of data on which it is trained to perform some functions and predictions. Specifically, in most cases AI has the advantage to make predictions much faster than the human brain and sometimes even providing observations that may not have been apparent to the human brain.

Also, Artificial intelligence has the ability to be utilized under difficult and dangerous conditions to jobs like mining, deep-sea and space exploration, military and rescue operations and many more. In addition, AI is also used for repetitive and exhausting tasks in many fields. For example, if a company has to deal daily with an enormous volume of data it is much more convenient to use a neural network which can work faster, continuously and with better precision. Some of these tasks are financial and scientific analysis, marketing research, environmental monitoring, fraud detection, supply chain management and many more.

So far, we have analysed the term AI and its applications but that is just the surface. A subcategory of AI is Machine learning which is also divided into four main types based on the learning style and the nature of data it has to process.

A) Supervised Learning

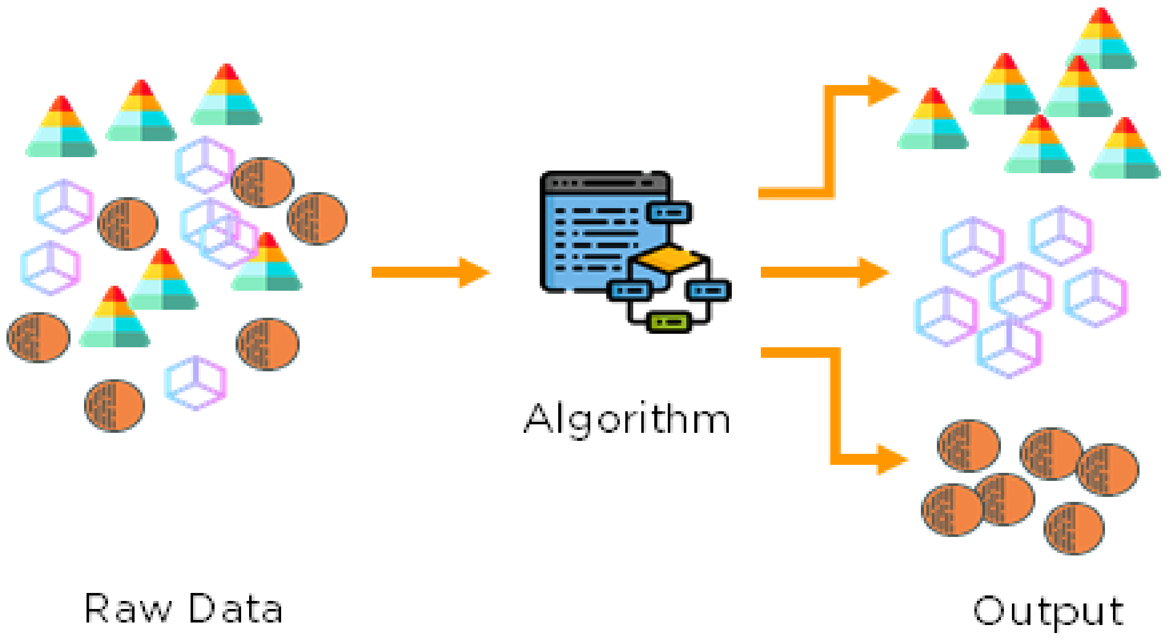
It is the most commonly used types of machine learning. In supervised learning, the algorithm is trained on a labelled dataset, where input data is paired with corresponding output labels, then the algorithm learns a mapping from inputs to outputs based on the provided labelled examples.



Supervised learning schema

B) Unsupervised Learning:

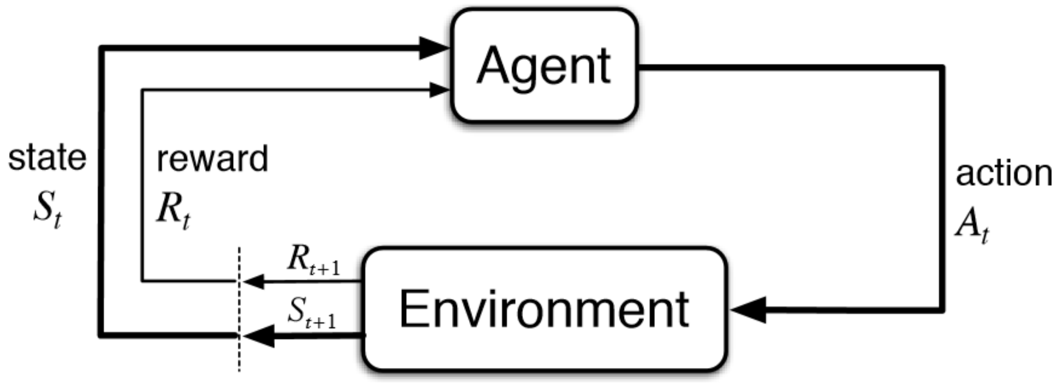
In contrast to supervised learning, Unsupervised involves training on unlabelled data and the goal is to find patterns or structures. within the data and without explicit output labels



2. Unsupervised learning schema

C) Reinforcement Learning:

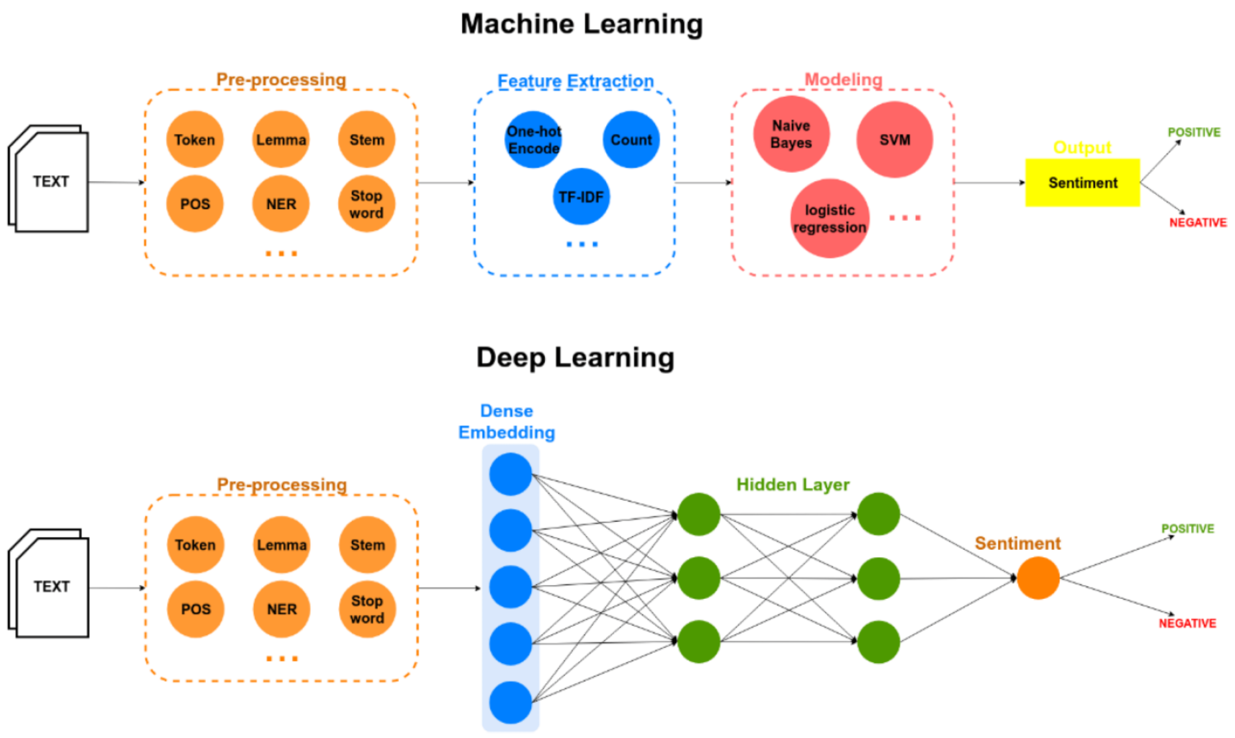
Based on trial-and-error approach, reinforcement learning involves training an agent to make decisions by interacting with an environment, receiving feedback in the form of rewards or penalties, while always trying to minimize the wrong steps. Reinforcement learning can be usually found in trading and finance applications and in automation, robotics and gaming industries.



3. Reinforcement learning schema.

D) Deep learning

A Deep neural network involves artificial neural networks with multiple layers, making it extremely powerful and that is why they are mostly used with large datasets and complex patterns. Specifically, deep learning is consisted by an input layer, hidden layers and an output layer.



4. Machine learning vs Deep learning schema

1.2 Introduction and description of the problem

This bachelor's thesis refers to the architecture of deep learning models. This is accomplished by using the deep learning models to classify tweets (Text Classification) related to Covid-19 to three different categories. The first category as tweets which are in favor of vaccination, agree and implement methods of protection against the virus. The second category as tweets which are against the vaccination and promote conspiracy theories. The third category as tweets which they have a neutral stance.

Specifically, to solve the problem above they have been collected a number of tweets. The dataset we are using is consisted by tweets from various datasets and a number of tweets with the use of Twitter API, we will refer to both ways of collecting data later in more detail.

The categorization has been accomplished with the help of NLP, a subfield of artificial intelligence which makes possible the communication between human and computer. NLP algorithms aims to understand the meaning of a text by extracting information from it, some applications are Sentiment Analysis, Text Summarization, Named Entity Recognition and Text Classification which is the one we are going to analyse.

For the implementation of Text Classification we used a Bert and a LSTM model. Bert stands for Bidirectional Encoder Representations from Transformers and it is cutting edge natural processing model developed by Google. Bert is a transformer pretrained using a combination of masked language modelling objective and next sentence prediction on a large corpus comprising the Toronto Book Corpus and Wikipedia.

The second model is a Bidirectional Long Short-Term Memory network an extension of a traditional LSTM model. It follows the RNN architecture by analysing data in both directions, from the beginning to the end and from the end to the beginning. In this manner the model will not make rush judgments.

In the following chapters we will further discuss about the creation of the models that we used, the collection and preparation of the data, and observations we made.

Chapter 2

AI as we already mention uses some types of models to make predictions, the most common of them are Naive Bayes, CNN, KNN, Random Forests, SVM, PCA, RNN and Transformers. The last two are the ones that we are going to analyze.

The history of transformers in the context of artificial intelligence (AI) is closely tied to the development of models like the Transformer architecture introduced by Vaswani et al. in the paper "Attention is All You Need," published in 2017. Transformers have since become a foundational technology in natural language processing (NLP) and other AI applications due to their ability to capture long-range dependencies and relationships in sequential data. Bert model is based on this technology.

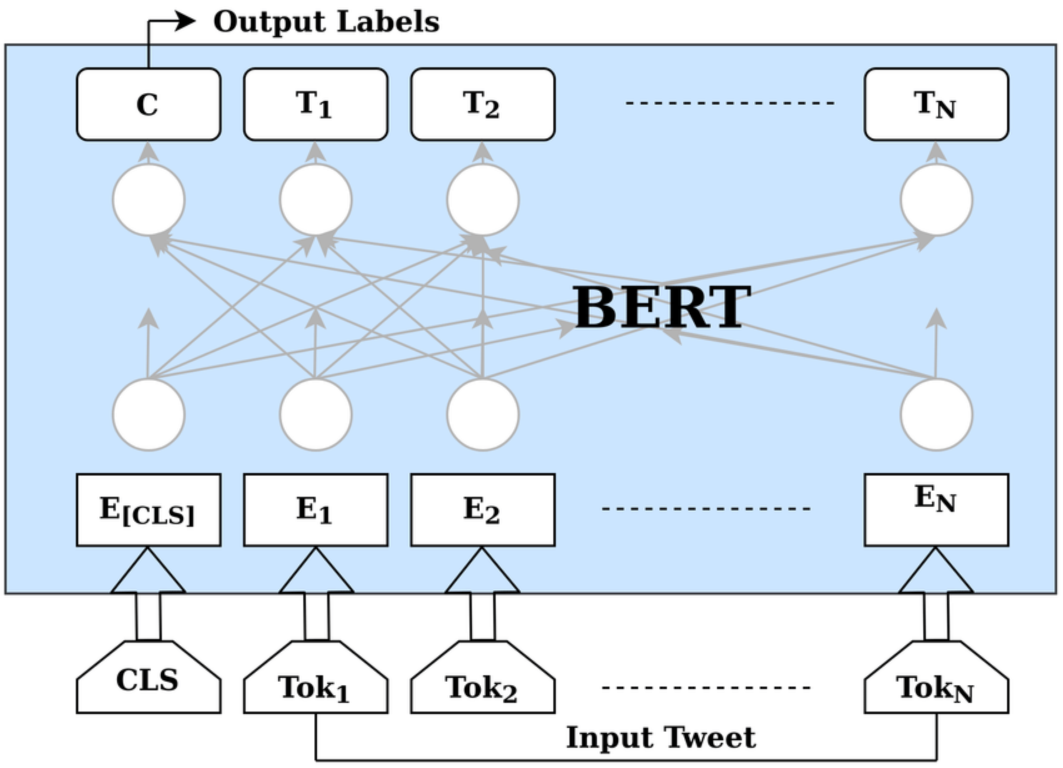
Recurrent Neural Networks (RNNs) are a class of neural networks designed for sequence data. They maintain hidden states to capture sequential dependencies. RNNs are suitable for tasks involving sequential information, such as natural language processing and time series prediction. However, they face challenges like vanishing gradients, limiting their ability to capture long-term dependencies, that means that the model is not able to keep information for a long time. To solve the underperformances that arise from the above issues, two neural network approaches are suggested, the gated recurrent unit (GRU) and the LSTM.

Some of the differences between those two architectures are that RNNs process sequences sequentially, struggling with long-term dependencies, while Transformers use parallelized self-attention mechanisms for simultaneous input processing. Transformers, including BERT, are more effective in capturing extensive context, allowing for better representation of sequential data. They overcome the limitations of RNNs, offering enhanced scalability and performance in natural language processing and other sequential tasks.

2.1 Bert model

The BERT model leverages transformer architecture for state-of-the-art natural language processing (NLP). Unlike traditional models, BERT processes text bidirectionally, considering both left and right context for each word.

Transformers employ a self-attention mechanism, allowing the model to weigh different parts of the input sequence, capturing intricate dependencies. BERT's pre-training involves unsupervised tasks—Masked Language Model and Next Sentence Prediction—on a vast corpus, imparting a rich understanding of language. Fine-tuning on specific tasks follows pre-training.

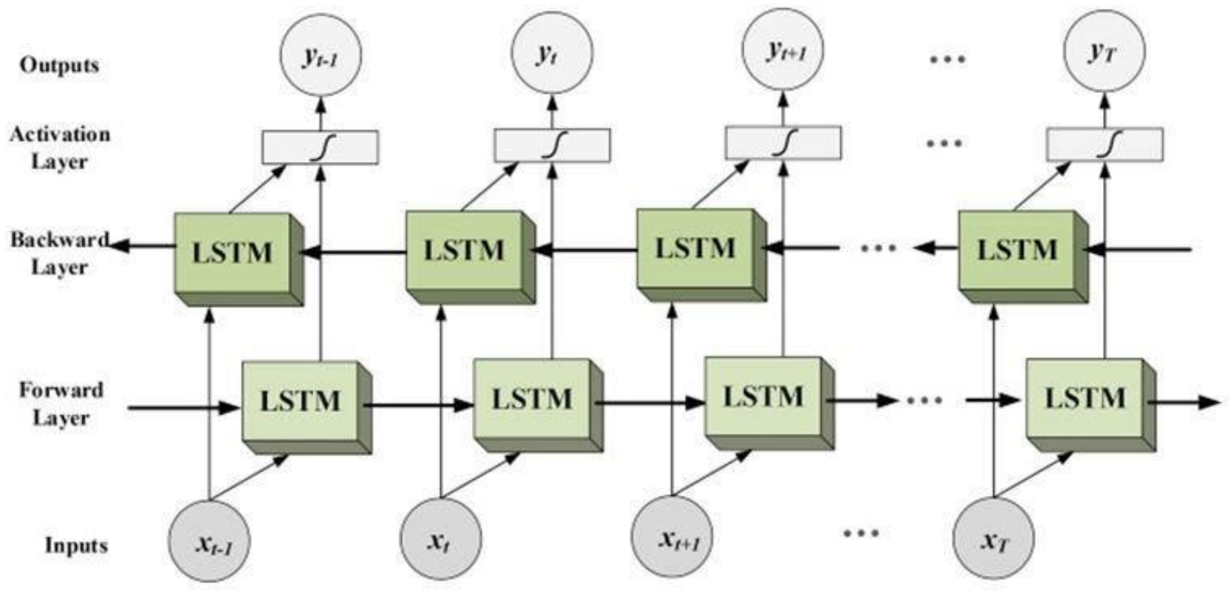


5. Bert Model Diagram

Tokenization and embedding into high-dimensional vectors enable BERT to generate contextualized representations, crucial for tasks where word meaning depends on context, these functions are going to be explained in detail. Multiple transformer layers and attention heads contribute to capturing hierarchical relationships and various abstraction levels. The contextualized embeddings and transfer learning from pre-training to downstream tasks empower BERT with a versatile understanding of language, resulting in remarkable performance across diverse NLP applications. The bidirectional and contextual nature of transformers in BERT represents a breakthrough, revolutionizing how models comprehend and process language for advanced NLP tasks.

2.2 Bidirectional LSTM

Bidirectional Long Short-Term Memory (BiLSTM) is a type of recurrent neural network (RNN) architecture designed to capture dependencies in sequential data more effectively. Unlike traditional LSTMs, which process data in a single direction, BiLSTM processes sequences in both forward and backward directions simultaneously. This bidirectional processing allows the network to capture context from both past and future inputs, enhancing its ability to understand and remember long range dependencies.



6. Bidirectional LSTM model diagram

As evident from the provided architecture, there are two LSTM models incorporated—one operating in the forward direction, while the other functions in the reverse direction. In a BiLSTM layer, there are two hidden states for each time step-one computed in the forward direction and the other in the backward direction. These hidden states are concatenated, providing a comprehensive representation of the input sequence. The bidirectional approach helps mitigate the vanishing gradient problem associated with standard LSTMs, enabling more efficient learning of long-term dependencies.

2.3 Natural Language Processing

Sometimes we are finding difficult to fully understand what a person says through human language with his voice or by text and that is because we are missing metaphors, idioms, sarcasm and plenty other factors. When an employee has to deal with too much data and extract information from them it can be time consuming and exhausting, making it easier for mistakes to be made. In addition, the need to analyse so many flowing data on the internet and extract a meaning from them which can a lead to scientific, financial, marketing or even fraud discovery is getting closer to impossible day by day.

That is how Natural Language Processing (NLP) is born, which is a branch of machine learning that empowers computers to understand, manipulate, and make sense of human language. NLP is increasingly integral to corporate solutions aimed at enhancing operational efficiency, boosting employee productivity, and simplifying vital business procedures. This is done with the use of NLP techniques or a combination of them like those below.

2.4 Text translation with BiLSTM and Bert model and their differences

Language translation involves complex syntactic and semantic understanding, where BiLSTM and BERT transformers play distinct roles. Bidirectional Long Short-Term Memory networks (BiLSTM) are recurrent neural networks adept at capturing sequential dependencies by processing input sequences bidirectionally. Their ability to retain context information makes them suitable for shorter sentences with local dependencies. However, BiLSTM may struggle with long-range dependencies and might not effectively capture intricate structures in language.

BERT (Bidirectional Encoder Representations from Transformers) transformers, on the other hand, use self-attention mechanisms for parallel processing, capturing global context efficiently. Pretrained on extensive corpora, BERT exhibits versatility but requires fine-tuning for specific tasks. It excels in understanding complex structures and long-range dependencies, making it well-suited for tasks involving nuanced language translation.

The choice between BiLSTM and BERT depends on the linguistic characteristics of the translation task. For simpler sentences with local dependencies, BiLSTM may suffice. In contrast, BERT transformers are preferred for tasks demanding a deeper comprehension of intricate linguistic structures and long-range dependencies. The decision hinges on the linguistic complexity and the nature of dependencies present in the data.

2.5 Text Summarization with BiLSTM and Bert model and their differences

The phrase “Time is gold”, is a very famous one but also a true one, especially when literally a business is losing money for the reason that it misses deadlines or even just because of underproductiveness. For example, employee to understand an undocumented project or an uncommented source code file is spending around 60% of his working time. Therefore, text summarization is coming to save the day.

Bidirectional Long Short-Term Memory networks present advantages and disadvantages in text summarization. On the positive side, BiLSTM networks excel at summarizing shorter texts with clear structures, providing interpretability, and demonstrating training efficiency. However, their limitations become apparent in handling long-range dependencies, and they may struggle with capturing complex semantic relationships in text in comparison to Bert. Despite their efficiency for certain tasks, BiLSTMs may fall short when faced with more intricate language patterns.

Utilizing BERT for text summarization offers substantial benefits, including a sophisticated understanding of context, improved comprehension of complex language structures, and exceptional performance on diverse language tasks that is why they have more versatility. However, the resource-intensive nature of BERT may pose challenges in terms of computational requirements and speed. Additionally, the model's extensive architecture might lead to limitations in real-time applications in which it is required a real-time quick response, for example in a chatbox situation.

2.6 Sentiment Analysis with BiLSTM and Bert model and their differences

Sentiment Analysis with Bidirectional Long Short-Term Memory has several advantages, including its ability to capture contextual information, discern intricate language nuances, and provide accurate sentiment classification. It excels in handling longer text sequences, making it suitable for diverse applications. However, BiLSTM's drawbacks include the need for substantial computational resources, potential overfitting on smaller datasets, and challenges in interpreting the model's decision-making process due to its complex architecture.

On the other hand, Sentiment Analysis with BERT offers unparalleled contextual understanding, outperforming traditional methods. Its pre-trained contextual embeddings capture intricate language nuances, enhancing sentiment interpretation. Additionally, BERT handles varying sentence lengths effectively. However, BERT's resource-intensive nature demands substantial computing power, limiting real-time applications. Its black-box nature poses challenges in model interpretability. Striking a balance between accuracy and computational efficiency is crucial when employing BERT.

2.7 Text Classification

Text classification is a natural language processing task that involves assigning predefined categories or labels to textual data. Without Text Classification some of the above NLP techniques could never work owing to the fact that it employs machine learning algorithms to autonomously assess and categorize text according to its content.

On top of that, Text classification is widely used in spam detection, sentiment analysis on social media, and document categorization. It brings automation to handle large volumes of text data, boosting efficiency in areas like customer support, content filtering, and information retrieval.

Based on Google developers, the Text Classification Workflow involves a systematic process to harness the power of machine learning for efficiently categorizing and analysing textual data :

Step 1: Gather Data

This initiates the process by collecting a diverse dataset representative of the problem at hand. For text classification, this would include a range of examples covering different classes or categories.

Step 2: Explore Your Data and choose a Model

Data exploration is crucial to understand patterns, distributions, and potential challenges within the dataset. Exploratory Data Analysis (EDA) aids in making informed decisions throughout subsequent steps. Selecting an appropriate model is a pivotal decision. For text classification, models like Naive Bayes, Support Vector Machines (SVM), or deep learning architectures such as recurrent neural networks (RNNs) and transformers like BERT are common choices.

Step 3: Prepare Your Data

Data preparation involves cleaning, tokenization, and vectorization of text. It transforms raw text into a format suitable for machine learning algorithms, converting words into numerical representations.

Step 4: Build, Train, and Evaluate Your Model

This step involves constructing the chosen model, training it on the prepared dataset, and evaluating its performance using metrics like accuracy, precision, recall, and F1-score.

Step 5: Tune Hyperparameters

Fine-tuning hyperparameters is crucial for optimizing model performance. This step involves adjusting settings like learning rates or regularization parameters to enhance the model's efficacy.

Step 6: Deploy Your Model

Once satisfied with the model's performance, deployment brings it into practical use. This could involve integration into applications, websites, or systems to perform real-time text classification tasks.

In essence, this workflow ensures a structured and iterative approach, emphasizing the importance of data quality, model selection, and continuous refinement to achieve robust text classification models. In the following chapters we will take a look at these step with the help of BiLSTM and Bert.

Chapter 3

3.1 The importance of a dataset

The foundation of a successful machine learning applications stands on a well-prepared collected dataset, so it can ensure that the models are trained on high-quality data, leading to better performances and enhanced reliability in real-world scenarios. Dataset types vary depending on the case, some of the most common are:

Numerical Data:

Example: Daily temperature recordings in Celsius, ranging from 15 to 30 degrees over a month, capturing variations in weather patterns.

Categorical Data:

Example: Movie genres (Action, Drama, Comedy) categorizing films based on their content, aiding in personalized recommendation systems for viewers.

Text Data:

Example: Customer reviews for a product, containing textual sentiments like "great," "average," or "poor," facilitating sentiment analysis for product feedback.

Image Data:

Example: Satellite images of urban areas, capturing visual patterns for land-use classification, assisting in urban planning and resource allocation. It is important to say that this type is also belonging to healthcare AI techniques, a very large area in which based on the patterns of medical images (e.g. X-rays) we can make diagnostics for patients.

Temporal Data:

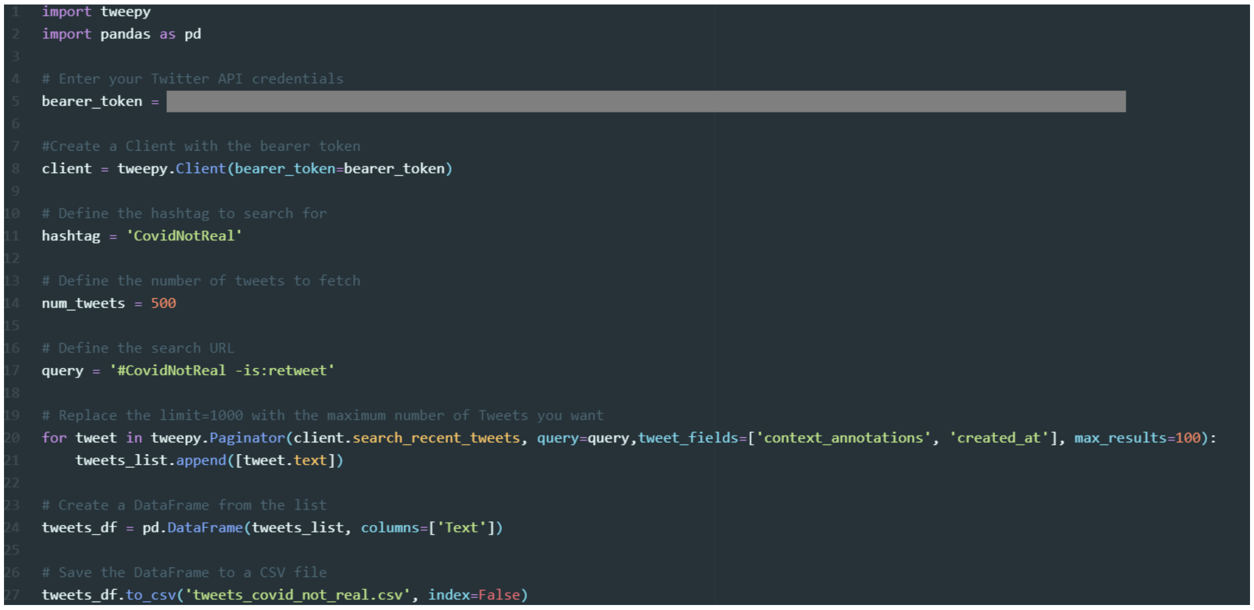
Example: Hourly energy consumption data, recording power usage fluctuations throughout a day and across seasons, enabling predictive modeling for energy demand and supply management.

These types of data are only some of the many and that is due machine learning continually evolves, introducing new types of datasets based on emerging technologies and research areas.

3.2 Dataset collection

Our Dataset has 2 columns, the first one is called “Tweets” which is consisted of 5860 tweets. Each one is manually categorized on the second column called “Type” based on its content.

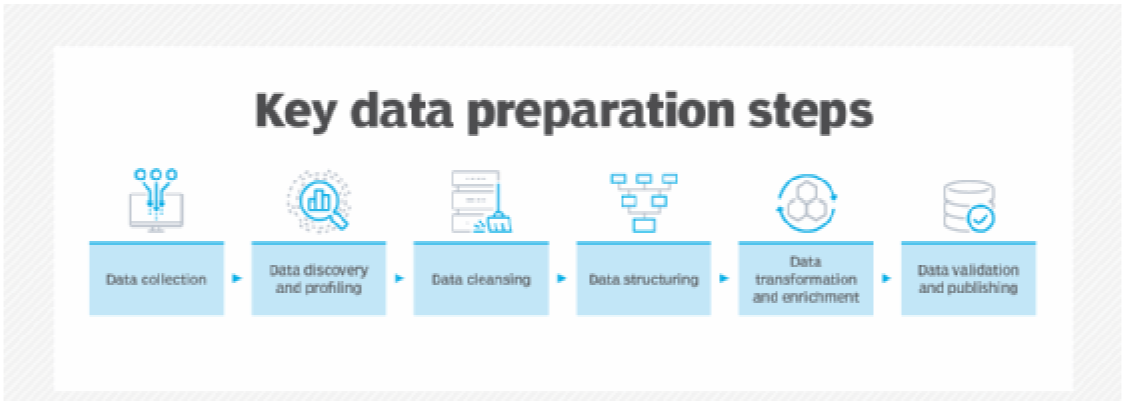
The 400 first tweets were collected totally manually by experimenting with Twitters search tools, so we can evolve a general idea around the content of the tweets, famous hashtags, trending words and social groups. Afterwords, the twitters API was used to collect around 2000 more tweets by referring to the hashtag that we wanted a tweet to have.



```python
import tweepy
import pandas as pd

# Enter your Twitter API credentials
bearer_token = [redacted]

#Create a Client with the bearer token
client = tweepy.Client(bearer_token=bearer_token)

# Define the hashtag to search for
hashtag = 'CovidNotReal'

# Define the number of tweets to fetch
num_tweets = 500

# Define the search URL
query = '#CovidNotReal -is:retweet'

# Replace the limit=1000 with the maximum number of Tweets you want
for tweet in tweepy.Paginator(client.search_recent_tweets, query=query,tweet_fields=['context_annotations', 'created_at'], max_results=100):
    tweets_list.append([tweet.text])

# Create a DataFrame from the list
tweets_df = pd.DataFrame(tweets_list, columns=['Text'])

# Save the DataFrame to a CSV file
tweets_df.to_csv('tweets_covid_not_real.csv', index=False)
```

7. Python script to extract tweets based on hashtags.

Sadly, after the acquisition of Twitter by Elon Musk on April 14, 2022, and concluded on October 27, 2022, the restricted free use of Twitter has been stopped. The need for more data leaded to search for covid related datasets on the internet, all of them are mentioned at the final chapter.

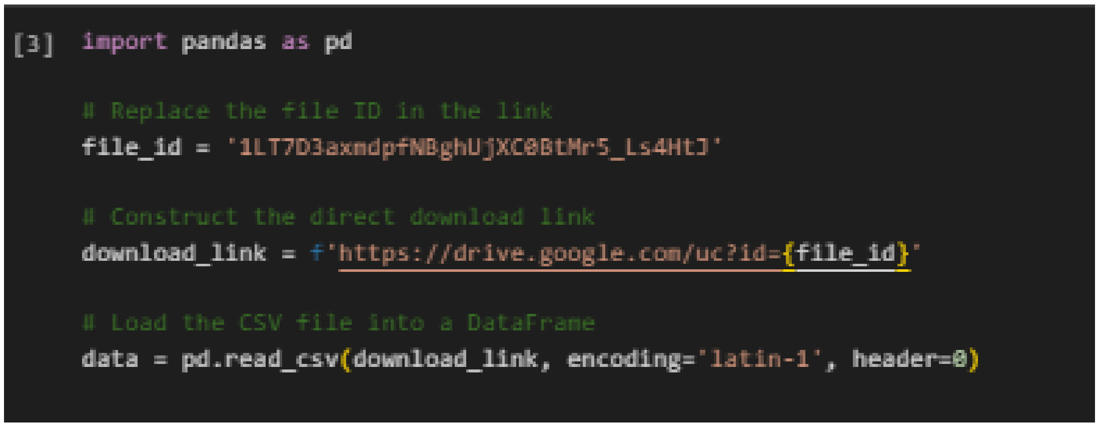

Key data preparation steps
Data collection
Data discovery and profiling
Data cleansing
Data structuring
Data transformation and enrichment
Data validation and publishing

8. Preparation steps

3.3 Dataset preparation

In this section we will describe in detail the google colab blocks of code that was used for the preparation of the dataset for Bert and BiLSTM. These two models have some common preparation steps but some other are different because of the different architecture.

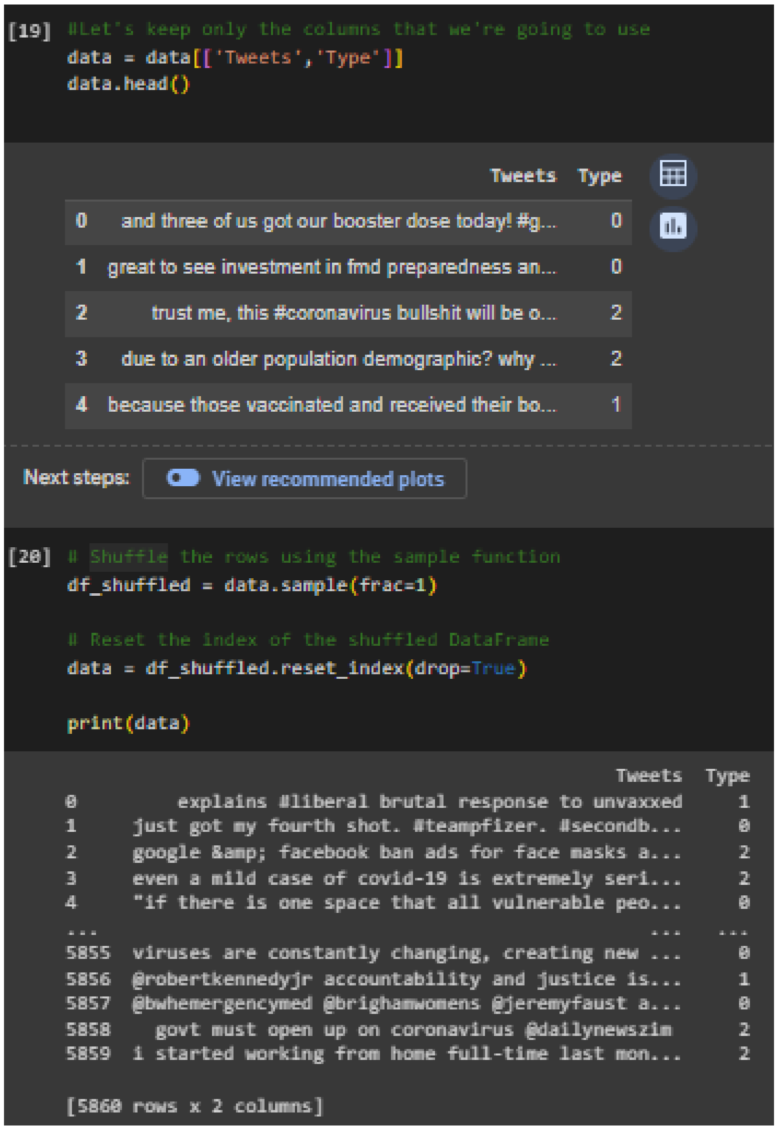
First, we load the csv file which contains the tweets via a google drive link with the help of pandas library. Pandas is a powerful data manipulation and analysis library for Python. It provides data structures like DataFrame for efficient data handling and analysis.



```
[3] import pandas as pd

    # Replace the file ID in the link
    file_id = '1LT7D3axmdpfNBghUjXC0BtMr5_Ls4HtJ'

    # Construct the direct download link
    download_link = f'https://drive.google.com/uc?id={file_id}'

    # Load the CSV file into a DataFrame
    data = pd.read_csv(download_link, encoding='latin-1', header=0)
```

9. Pandas import and dataset load

At the image below then we shuffle the rows but also keeping the type of each tweet at the second column. The reason behind this is to ensure that the training is not biased, especially during the split between training and testing that we will discuss later on. We are keeping the columns that we are going to need for the training, only the Tweets and their Type.



```
[19] #Let's keep only the columns that we're going to use
     data = data[['Tweets','Type']]
     data.head()
```

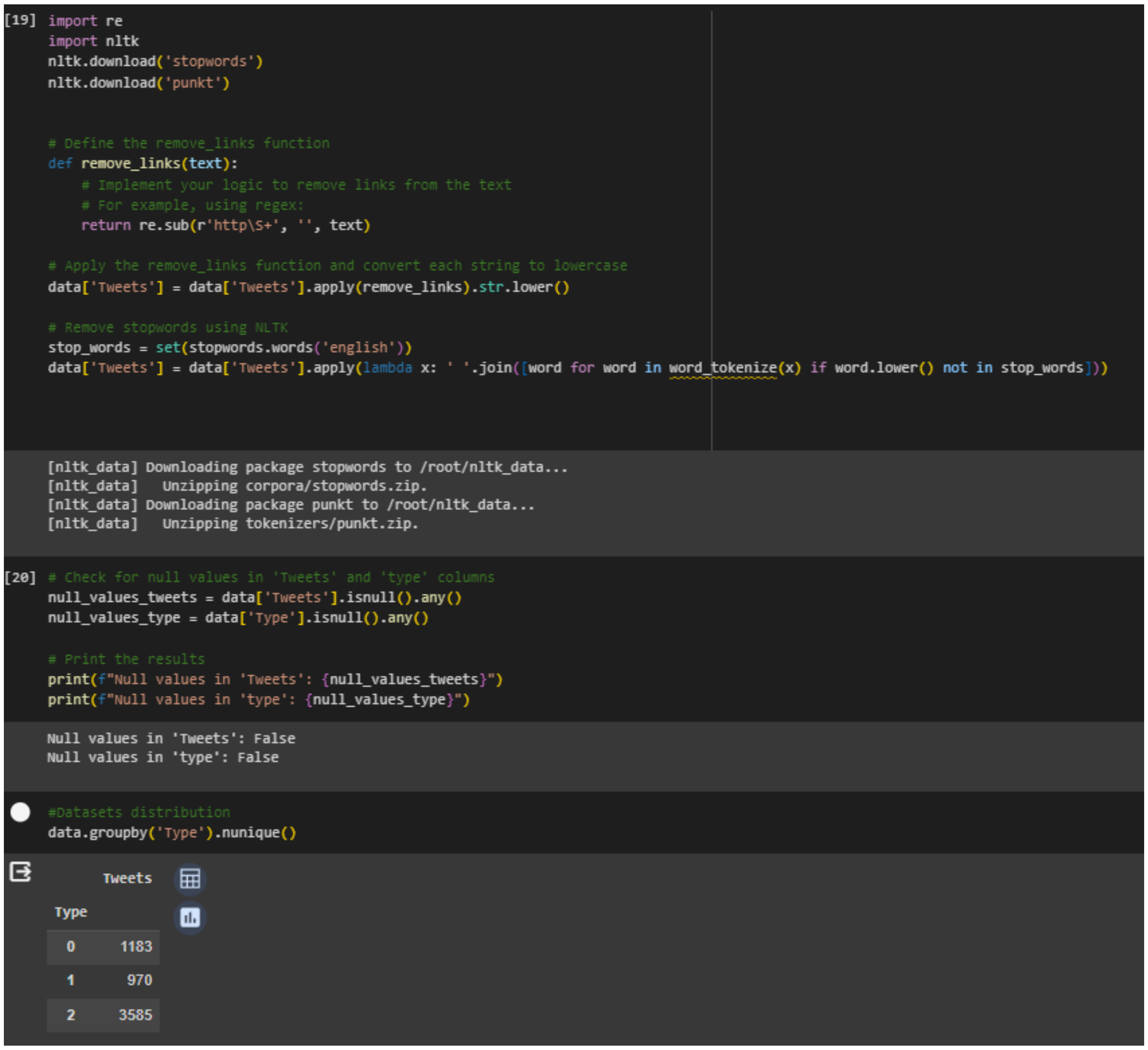
```
                                         Tweets  Type
0    and three of us got our booster dose today! #g...   0
1   great to see investment in fmd preparedness an...   0
2         trust me, this #coronavirus bullshit will be o...   2
3    due to an older population demographic? why ...   2
4   because those vaccinated and received their bo...   1
```

```
[20] # Shuffle the rows using the sample function
     df_shuffled = data.sample(frac=1)

     # Reset the index of the shuffled DataFrame
     data = df_shuffled.reset_index(drop=True)

     print(data)
```

```
                                                 Tweets  Type
0            explains #liberal brutal response to unvaxxed     1
1        just got my fourth shot. #teampfizer. #secondb...     0
2        google &amp; facebook ban ads for face masks a...     2
3        even a mild case of covid-19 is extremely seri...     2
4        "if there is one space that all vulnerable peo...     0
...                                                    ...   ...
5855  viruses are constantly changing, creating new ...     0
5856  @robertkennedyjr accountability and justice is...     1
5857  @bwhemergencymed @brighamwomens @jeremyfaust a...     0
5858    govt must open up on coronavirus @dailynewszim     2
5859  i started working from home full-time last mon...     2

[5860 rows x 2 columns]
```

10. Dataset format

In addition, a function is created which is used to remove any links from the tweets that could confuse our two models, making them more complex by thinking that every link is a different word. Also, we are converting every upper-case letter to a lower case letter, so that a word like “Deathshot” to be the same and have the same impact like the word “deathshot”. At the end with the NLTK dataset some English stopwords are excluded from the Tweets.



```
[19] import re
     import nltk
     nltk.download('stopwords')
     nltk.download('punkt')


     # Define the remove_links function
     def remove_links(text):
         # Implement your logic to remove links from the text
         # For example, using regex:
         return re.sub(r'http\S+', '', text)

     # Apply the remove_links function and convert each string to lowercase
     data['Tweets'] = data['Tweets'].apply(remove_links).str.lower()

     # Remove stopwords using NLTK
     stop_words = set(stopwords.words('english'))
     data['Tweets'] = data['Tweets'].apply(lambda x: ' '.join([word for word in word_tokenize(x) if word.lower() not in stop_words]))
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

```
[20] # Check for null values in 'Tweets' and 'type' columns
     null_values_tweets = data['Tweets'].isnull().any()
     null_values_type = data['Type'].isnull().any()

     # Print the results
     print(f"Null values in 'Tweets': {null_values_tweets}")
     print(f"Null values in 'type': {null_values_type}")
```

```
Null values in 'Tweets': False
Null values in 'type': False
```

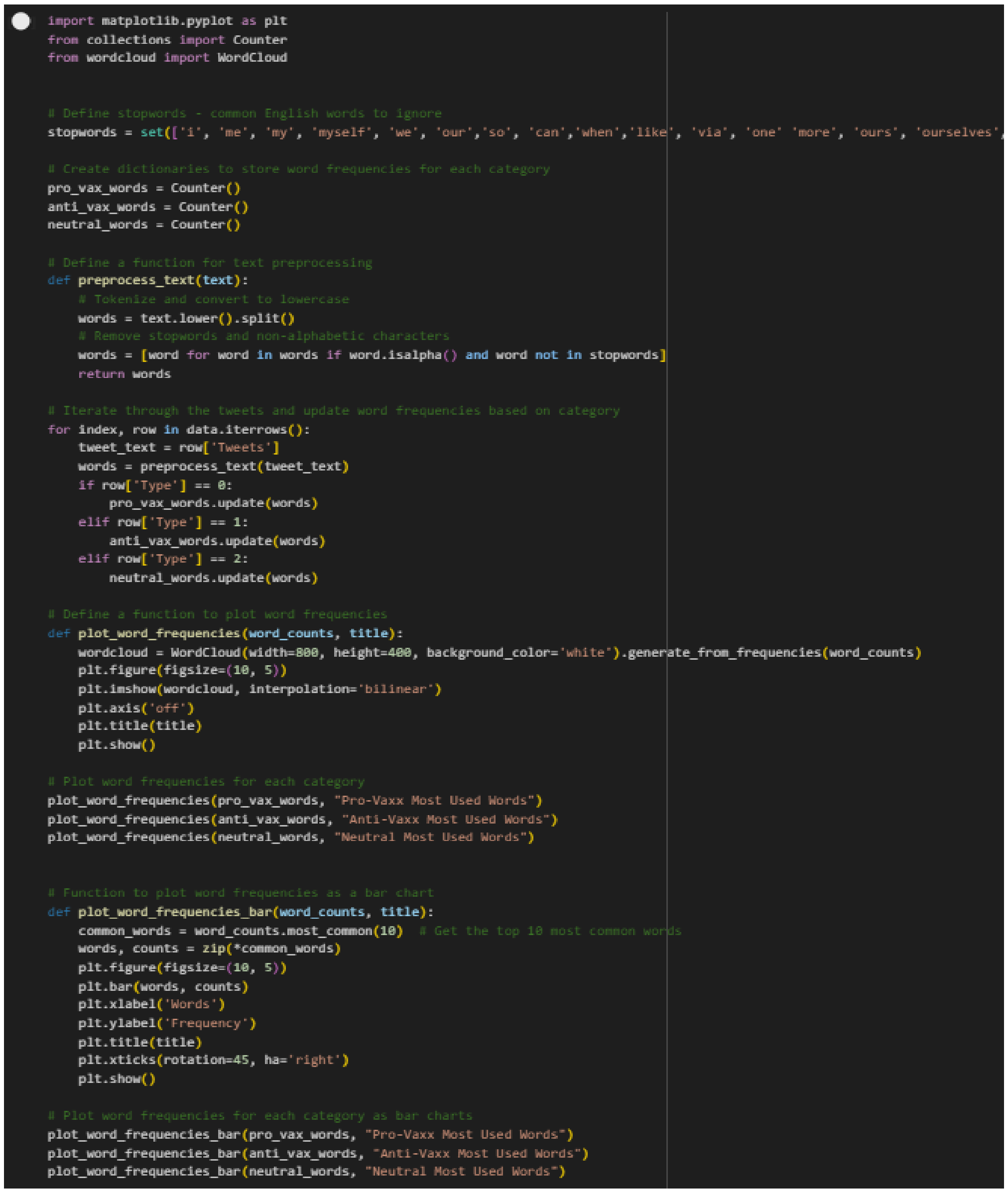
```
#Datasets distribution
data.groupby('Type').nunique()
```

	Tweets
Type	
0	1183
1	970
2	3585

11. Cleaning data and checking for null values

3.4 Graphs and Charts

Moreover, we are gaining some insights into the most frequently used word within each category (pro-vaccine, anti-vaccine, or neutral) and visualize them through word clouds and bar charts with the help of Matplotlib, Counter and WorldCloud libraries.

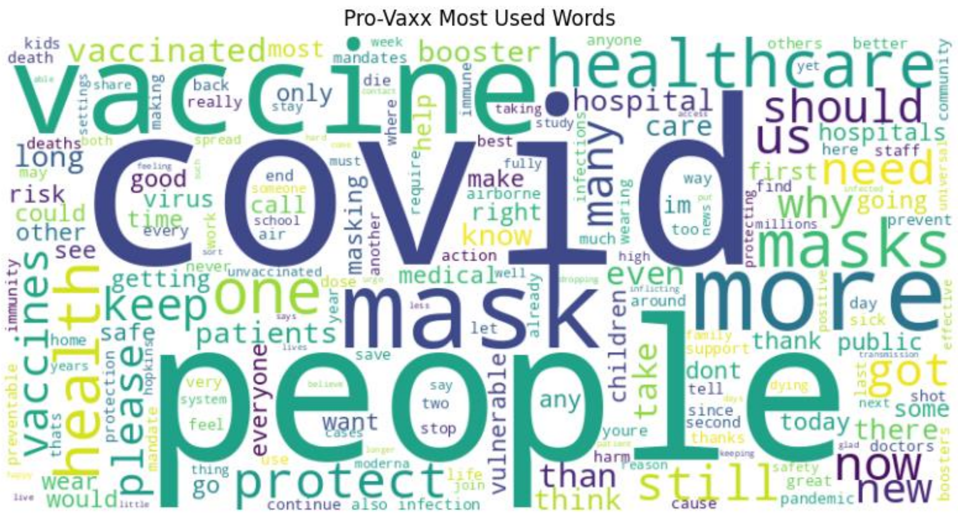


```python
import matplotlib.pyplot as plt
from collections import Counter
from wordcloud import WordCloud


# Define stopwords - common English words to ignore
stopwords = set(['i', 'me', 'my', 'myself', 'we', 'our','so', 'can','when','like', 'via', 'one' 'more', 'ours', 'ourselves',

# Create dictionaries to store word frequencies for each category
pro_vax_words = Counter()
anti_vax_words = Counter()
neutral_words = Counter()

# Define a function for text preprocessing
def preprocess_text(text):
    # Tokenize and convert to lowercase
    words = text.lower().split()
    # Remove stopwords and non-alphabetic characters
    words = [word for word in words if word.isalpha() and word not in stopwords]
    return words

# Iterate through the tweets and update word frequencies based on category
for index, row in data.iterrows():
    tweet_text = row['Tweets']
    words = preprocess_text(tweet_text)
    if row['Type'] == 0:
        pro_vax_words.update(words)
    elif row['Type'] == 1:
        anti_vax_words.update(words)
    elif row['Type'] == 2:
        neutral_words.update(words)

# Define a function to plot word frequencies
def plot_word_frequencies(word_counts, title):
    wordcloud = WordCloud(width=800, height=400, background_color='white').generate_from_frequencies(word_counts)
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')
    plt.title(title)
    plt.show()

# Plot word frequencies for each category
plot_word_frequencies(pro_vax_words, "Pro-Vaxx Most Used Words")
plot_word_frequencies(anti_vax_words, "Anti-Vaxx Most Used Words")
plot_word_frequencies(neutral_words, "Neutral Most Used Words")


# Function to plot word frequencies as a bar chart
def plot_word_frequencies_bar(word_counts, title):
    common_words = word_counts.most_common(10)  # Get the top 10 most common words
    words, counts = zip(*common_words)
    plt.figure(figsize=(10, 5))
    plt.bar(words, counts)
    plt.xlabel('Words')
    plt.ylabel('Frequency')
    plt.title(title)
    plt.xticks(rotation=45, ha='right')
    plt.show()

# Plot word frequencies for each category as bar charts
plot_word_frequencies_bar(pro_vax_words, "Pro-Vaxx Most Used Words")
plot_word_frequencies_bar(anti_vax_words, "Anti-Vaxx Most Used Words")
plot_word_frequencies_bar(neutral_words, "Neutral Most Used Words")
```

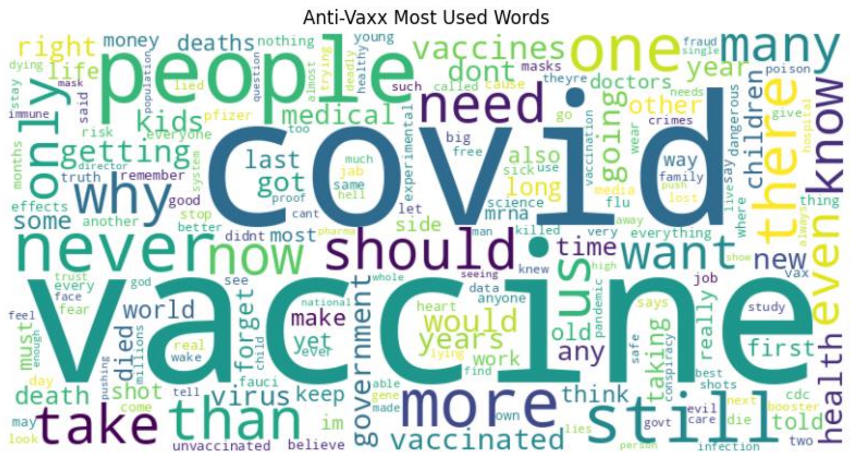
12. Creating graphs and charts

The block of code begins by defining a common set of English words to exclude them from the analysis, so that they will not interfere to the statistics. Afterwards, it uses the Counter dictionaries to store word frequencies for each category, the text processing function "preprocess_text" tokenizes and removes non-alphabetic characters. Then through the loop the frequency of each word for each category is being updated so that we can generate a cloud of the most used words with WordCloud.

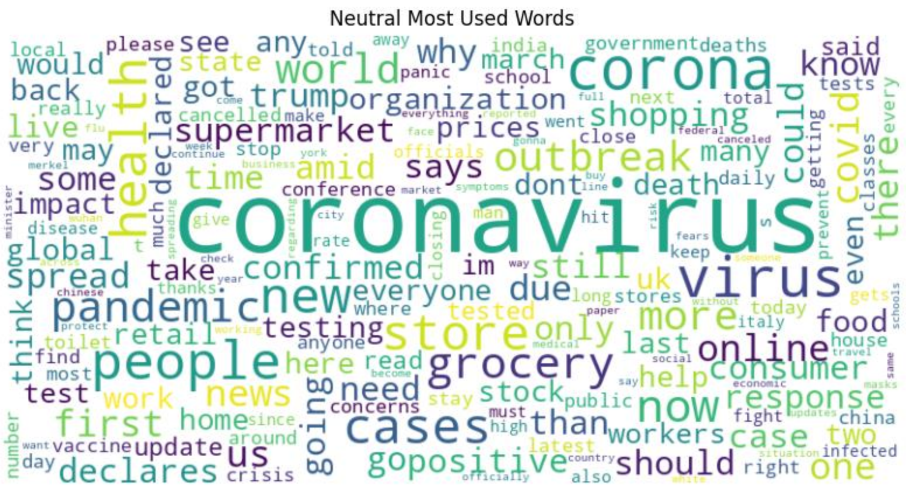
The plot_word_frequencies function is designed to display word frequencies in a visually appealing manner:



13. Word Frequency for Pro-Vaxx Tweets

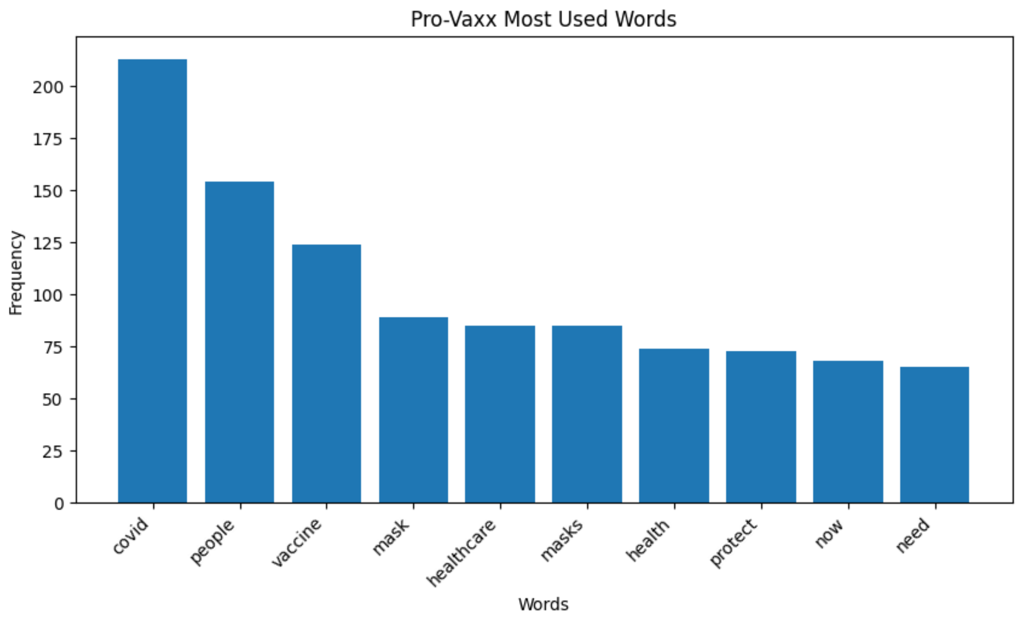


14. Word Frequency for Anti-Vaxx Tweets

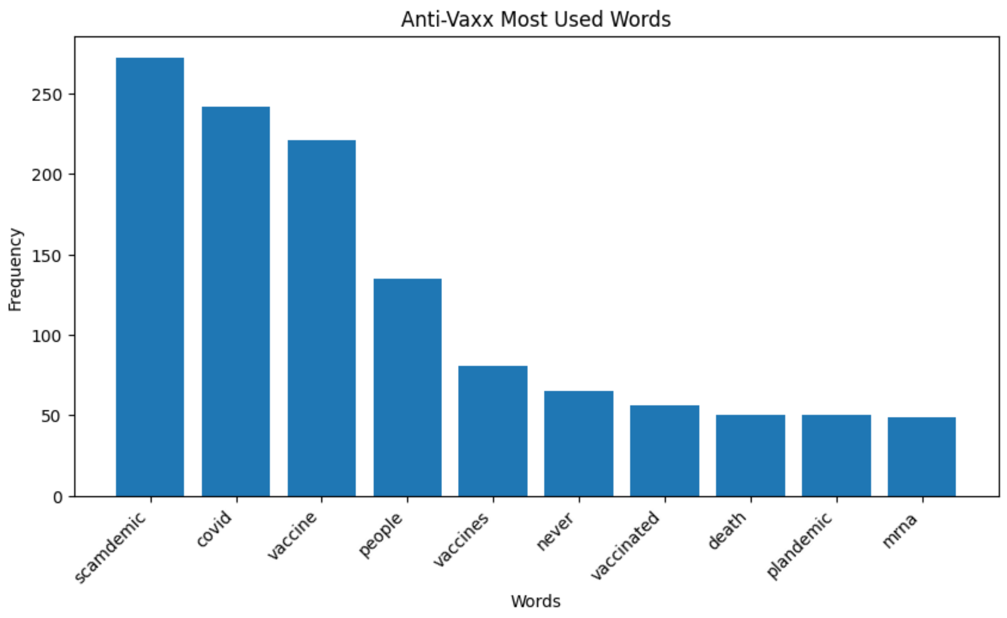


15. Word Frequency for Neutral Tweets

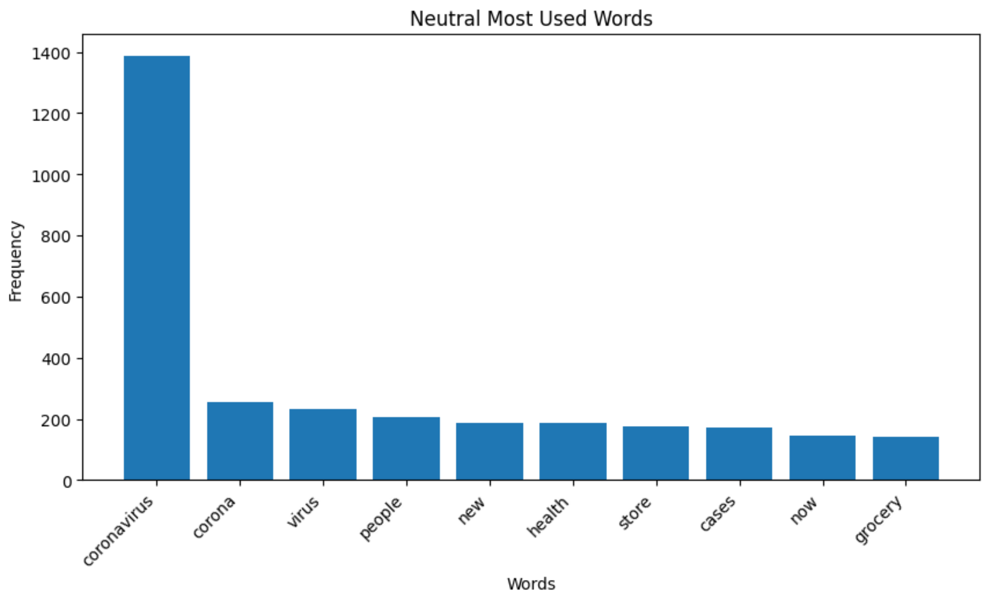
Additionally, the script includes a function, plot_word_frequencies_bar to create bar charts showing the top 10 most common words for each category:



16. Word Frequency for Pro-Vaxx Tweets with custom chart



17. Word Frequency for Anti-Vaxx Tweets with custom chart



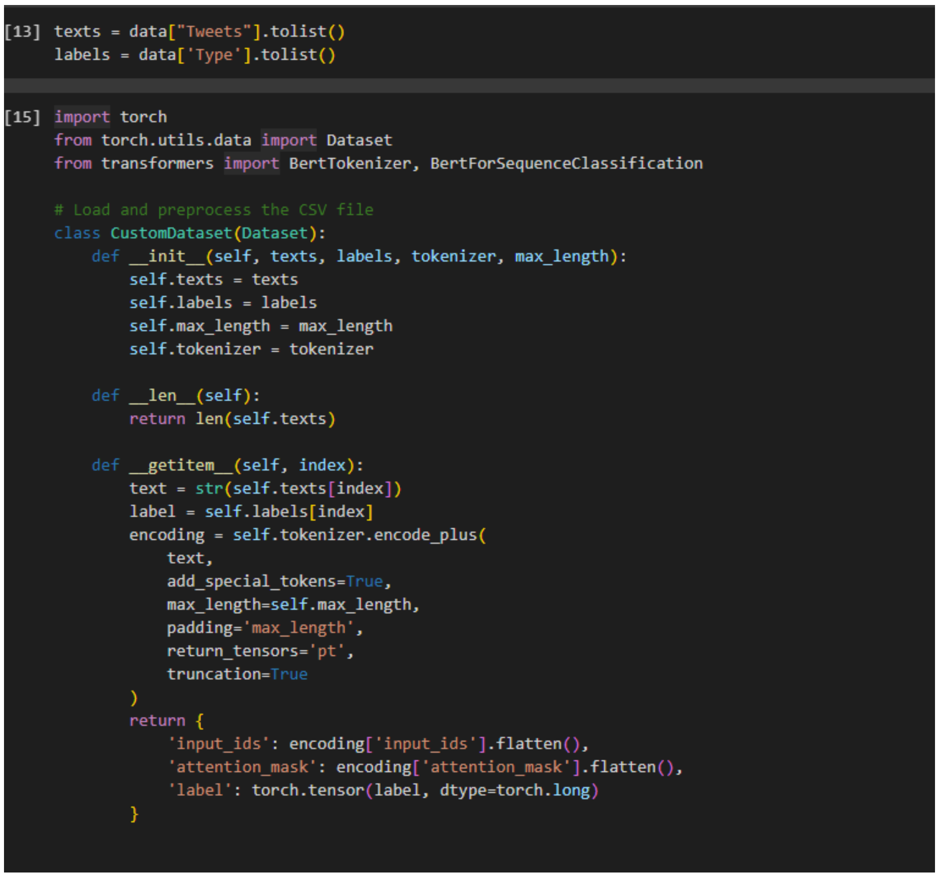
18. Word Frequency for Neutral Tweets with custom chart

These visualizations are important to understand the content of a dataset and valuable insights by extracting useful information like patterns and biased words.

3.5 Dataset preparation for Bert Model

Till now we are utilizing some cleaning techniques so we can be more specific and less bewilder for what we want, having as an outcome to get as much precise results as we can. Nevertheless, the data preparation does not stop here, it is mandatory to change the form of the dataset accordingly, so that each model can understand the human language as we have previously said.

For the Bert model, we must follow the following conventions:



```
[13] texts = data["Tweets"].tolist()
     labels = data['Type'].tolist()

[15] import torch
     from torch.utils.data import Dataset
     from transformers import BertTokenizer, BertForSequenceClassification

     # Load and preprocess the CSV file
     class CustomDataset(Dataset):
         def __init__(self, texts, labels, tokenizer, max_length):
             self.texts = texts
             self.labels = labels
             self.max_length = max_length
             self.tokenizer = tokenizer

         def __len__(self):
             return len(self.texts)

         def __getitem__(self, index):
             text = str(self.texts[index])
             label = self.labels[index]
             encoding = self.tokenizer.encode_plus(
                 text,
                 add_special_tokens=True,
                 max_length=self.max_length,
                 padding='max_length',
                 return_tensors='pt',
                 truncation=True
             )
             return {
                 'input_ids': encoding['input_ids'].flatten(),
                 'attention_mask': encoding['attention_mask'].flatten(),
                 'label': torch.tensor(label, dtype=torch.long)
             }
```

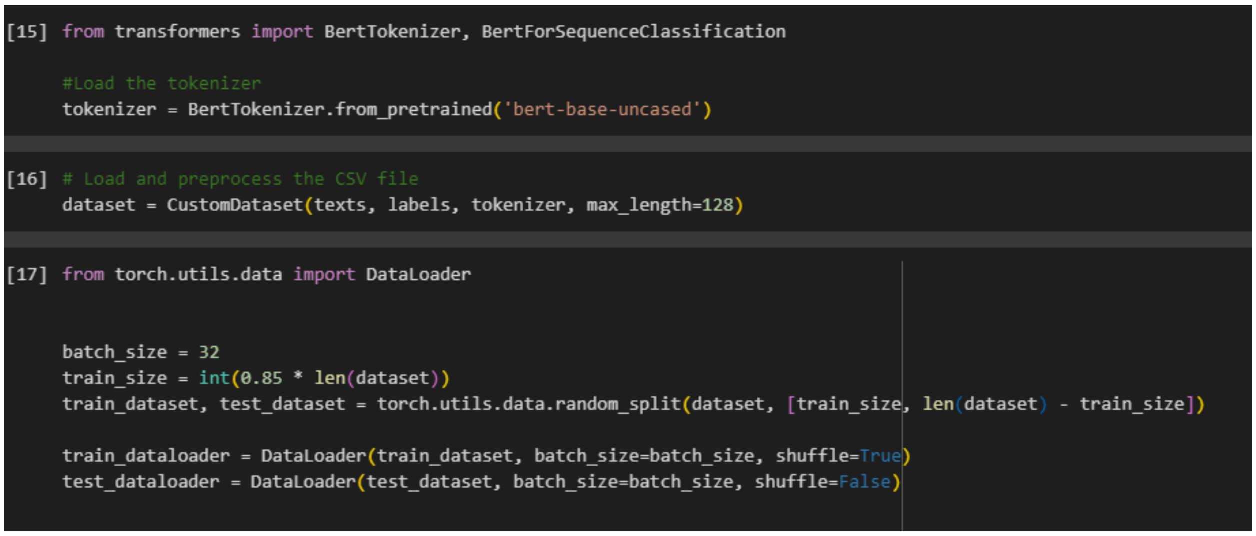
19. Dataset preparation for the Bert model 1.1

First of all, the columns “Tweets” and “Type” are moved to two lists accordingly. Afterwards, torch library is imported which is the core PyTorch library for tensor computations and deep learning. The importation of Dataset from torch.utils.data is a data handling module, it provides the ability to create custom datasets, which in our case it is going to handle and change data efficiently for the training part of the model.

Next, we create a class which defines a custom dataset that inherits from the Dataset class which is provided from the Pytorch library. Then with the function “ __init__ “, we initialize and below we assing the dataset with the provided texts, labels, a tokenizer and the maximum length for the text encoding.

The function “__len__” is used to return the text length of all the tweets. With the “__getitem__” method we process a specific item in a dataset, it takes an index and retrieves the corresponding text and label. The text is tokenized using a tokenizer, ensuring a maximum length, and adding special tokens. The resulting encoding includes input IDs and attention mask tensors. The method returns a dictionary containing flattened input IDs and attention mask, along with the label converted to a PyTorch tensor with a long data type. This facilitates seamless integration with PyTorch for natural language processing tasks, where input data needs to be transformed into a format suitable for training machine learning models.

In addition, from the Hugging Face Transformers library we are utilizing the BertTokenizer, it is used for tokenizing and preprocessing text data for our Bert model, meaning that it will convert our tweets into numerical tokens that will be inserted into our Bert model. At the next lines, we use the libraries to load a pre-trained BERT model and its associated tokenizer.


```
[15] from transformers import BertTokenizer, BertForSequenceClassification

     #Load the tokenizer
     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

[16] # Load and preprocess the CSV file
     dataset = CustomDataset(texts, labels, tokenizer, max_length=128)

[17] from torch.utils.data import DataLoader


     batch_size = 32
     train_size = int(0.85 * len(dataset))
     train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, len(dataset) - train_size])

     train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
     test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

20. Dataset preparation for the Bert model.1.2

The BertTokenizer.from_pretrained('bert-base-uncased') initializes the tokenizer for the 'bert-base-uncased' model, which is trained on uncased English text. Next, we are creating an instance of the CustomDataset class, this line initializes the dataset using the following parameters:

texts: the collected tweets.

labels: The corresponding labels for the text data.tokenizer max_length=128: A specified maximum length for the input sequences. This parameter is used for padding or truncating the input sequences to a consistent length.The CustomDataset class is expected to handle the loading and preprocessing of the dataset based on these parameters, making it suitable for training our Bert model.

Later on, another important class we are importing from the PyTorch library is Dataloader, it facilitates loading and batching of data during training and testing. In our case we are setting the batch size at 32 size, it means that 32 data samples are going to be processed in each iteration during the training.
Then we with the “train_size = int(0.85 * len(dataset))”, we indicate that the size of the training set is 85% of the total dataset length. Right after, we split the dataset into training and testing randomly to be sure that both datasets have diversity.

When all was said and done, the data preparation for the Bert model ends by creating a dataloader for the training and test by loading their corresponding datasets to them, the same batch size but enabling the shuffling the data in each epoch only for the “train_dataloader”.

3.6 Dataset preparation for BiLSTM Model

Just like Bert model, before the training of the BiLSTM model the data have to be converted to the right form. In the realm of natural language processing, the efficacy of deep learning models hinges on the quality of their training datasets. Crafting an optimal dataset for a Bidirectional

Long Short-Term Memory (BiLSTM) model involves a delicate orchestration of textual data preprocessing. We will dive into the thorough steps taken to prepare a robust dataset, encompassing tokenization, sequence padding, and the integration of pre-trained GloVe embeddings. Each facet of this process, outlined in the subsequent chapters, contributes to the foundation upon which the BiLSTM model will embark on its journey to unravel the intricacies of language and meaning.

First of all, we are setting max_words to the total number of texts to represent the maximum number of words considered during tokenization.

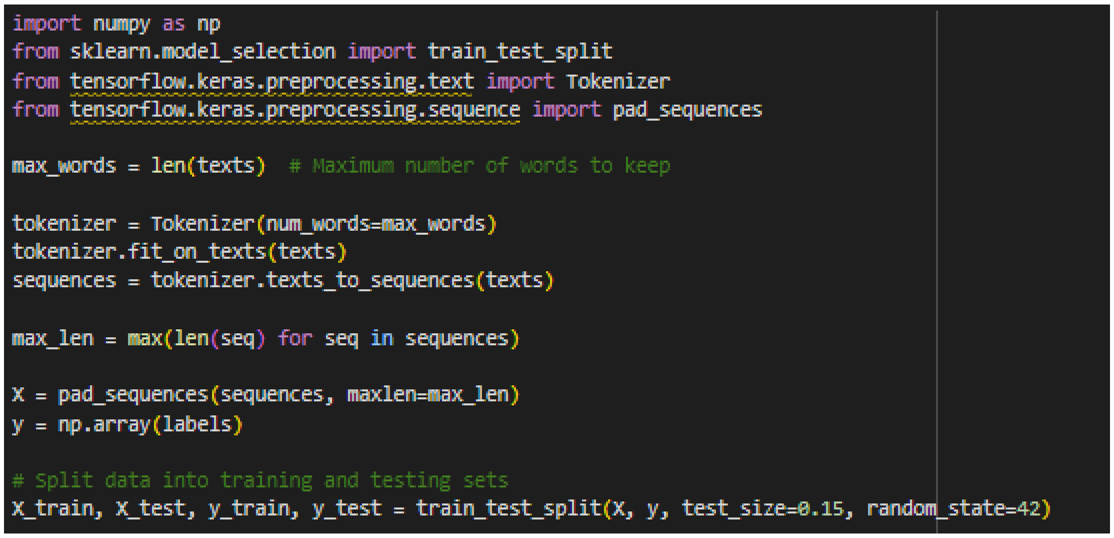
Then, just like Bert model we initialize the 'Tokenizer', which will be used to vectorize text. num_words=max_words limits the tokenizer to consider only the top max_words most frequent words in the dataset. Afterwards, we create a dictionary of word indices based on word frequency. It converts each word in the texts into a unique integer index. At the exact next line, the transformation of text into sequences of integers is followed, where each integer corresponds to the index of a word in the tokenizer's dictionary. This makes the text data suitable for input into machine learning models.

Later, "max_len" is calculated and defined by the length of the longest sequence in the sequences list. This value is used to pad all sequences to the same length for uniform input size in a model. We also define a variable ('X') ,which represents padded data of each sequence to ensure that they all have the same length, defined by max_len.

If a sequence is shorter than max_len, it is padded with zeros (or another specified value) at the beginning or end, If it's longer, it's truncated.

In another variable ('y') we also define the list of our labels into a Numpy array, so they can be used in our machine learning framework.

For the crucial step of the dataset splitting, we use the ‘train_test_split’ function, which is imported from scikit-learn, enabling the convenient division of a dataset into training and testing sets for machine learning model development. More specifically, X and y are divided such that 85% of the data goes into the training set (X_train, y_train) and 15% goes into the testing set (X_test, y_test), as specified by test_size=0.15. Last but not least we set a random_state to 42, which ensures reproducibility by seeding the random number generator, so the split is the same each time the code is run.



```
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

max_words = len(texts)  # Maximum number of words to keep

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

max_len = max(len(seq) for seq in sequences)

X = pad_sequences(sequences, maxlen=max_len)
y = np.array(labels)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)
```

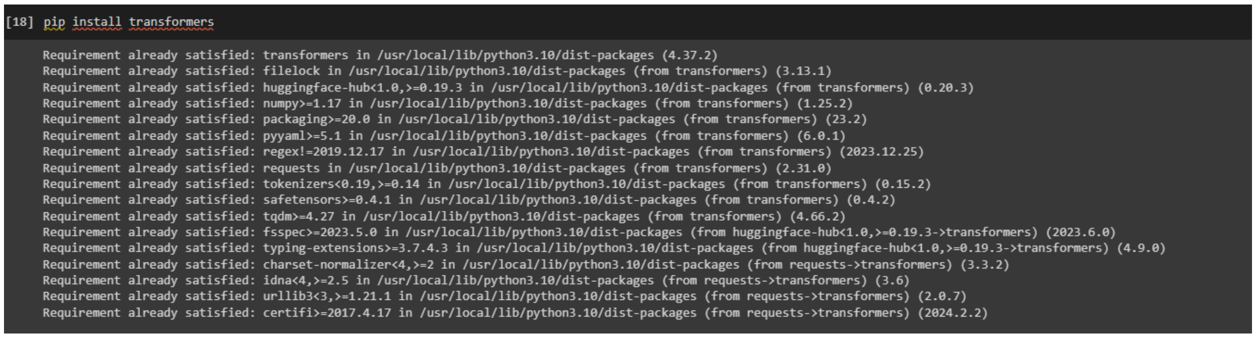
21. Dataset preparation for the BiLSTM model,

Chapter 4

4.1 Bert model creation

In the journey toward building a powerful sentiment analysis model, we now embark on the creation of a BERT model. The data, meticulously prepared in the earlier phase, lays the foundation for our BERT model. This transformer-based architecture, pre-trained on vast corpora, brings unparalleled proficiency in capturing intricate patterns and nuances of language. As we delve into model construction, the utilization of fine-tuning on natural language analysis, allows BERT to adapt and excel in our domain. Our arsenal includes the AdamW optimizer, a variant of Adam designed for BERT fine-tuning, and the indispensable learning rate scheduler to optimize training efficiency. We embrace the flexibility of dropout layers to prevent overfitting, enhancing the model's generalization capabilities. In this journey, each line of code becomes a keystroke in orchestrating the symphony of machine learning. The model evolves through epochs, refining its comprehension of sentiment nuances with every iteration. As we navigate this intricate path, the convergence of data preparation and BERT model creation promises a sentiment analysis model that transcends boundaries, unlocking the profound subtleties embedded in language.

We start by installing to our environment the transformers, the pip install transformers command is crucial for training BERT (Bidirectional Encoder Representations from Transformers) models as it installs the "transformers" library, a key dependency for working with transformer-based models like BERT. This library provides pre-trained transformer models, including BERT, and enables efficient implementation, fine-tuning, and utilization of these models for various natural language processing tasks. By installing this package, users gain access to a wealth of pre-trained models, tokenizers, and utilities, streamlining the development and training of BERT-based models for our corresponding task.

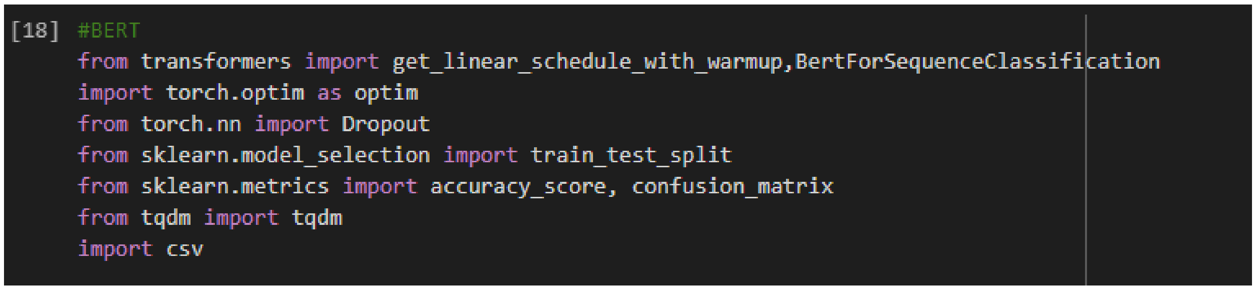


```
[18] pip install transformers

     Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-packages (4.37.2)
     Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from transformers) (3.13.1)
     Requirement already satisfied: huggingface-hub<1.0,>=0.19.3 in /usr/local/lib/python3.10/dist-packages (from transformers) (0.20.3)
     Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (1.25.2)
     Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers) (23.2)
     Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (6.0.1)
     Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (2023.12.25)
     Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers) (2.31.0)
     Requirement already satisfied: tokenizers<0.19,>=0.14 in /usr/local/lib/python3.10/dist-packages (from transformers) (0.15.2)
     Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (0.4.2)
     Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-packages (from transformers) (4.66.2)
     Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.19.3->transformers) (2023.6.0)
     Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.19.3->transformers) (4.9.0)
     Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.3.2)
     Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.6)
     Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.7)
     Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2024.2.2)
```

22. Installation of transformers,

4.2 Bert model initializations

Then we are importing the required functions for the operation of the training. Firstly we import from the transformers library the "BertForSequenceClassification", which is a fine-tuned pre-trained Bert model for sequence classification tasks and then the "get_linear_schedule_with_warmup" function, it's purpose is to schedule the learning rate during training, crucial for fine-tuning pre-trained models like BERT, ensuring effective optimization. Afterwards, we import “optim” from torch library which provides optimization algorithms, AdamW, a variant of Adam with weight decay, suitable for BERT fine-tuning. From the same library we continue by importing the “Dropout” function, responsible for the regularization of our model, by randomly setting a fraction of input units to zero during training, preventing overfitting.



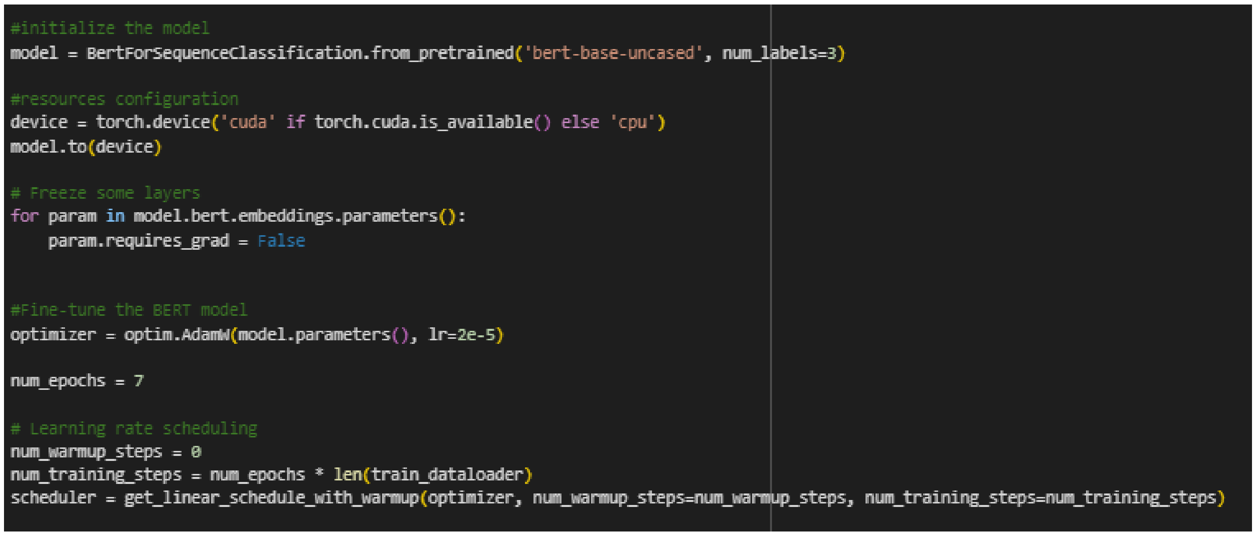
```
[18] #BERT
     from transformers import get_linear_schedule_with_warmup,BertForSequenceClassification
     import torch.optim as optim
     from torch.nn import Dropout
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import accuracy_score, confusion_matrix
     from tqdm import tqdm
     import csv
```

23. Function importation for Bert model

Next, from “sklearn” library we import three functions. The first is the “train_test_split”, which splits data into training and testing sets, facilitating model evaluation on unseen data for robustness assessment. Then, from “sklearn.metrics” we import the “accuracy_score” function which calculates the accuracy of model predictions, comparing them to true labels and the “confusion_matrix” function, which constructs a confusion matrix to evaluate the performance of a classification algorithm by visualizing true positives.

At the end, we import the “tqdm” for visual needs, which provides a progress bar during the training. The last is the “csv” function, which allows us to handle csv files during training.

The building of the model is starting by initializing the model with BertForSequenceClassification.from_pretrained ('bert-base-uncased', num_labels=3) loads the BERT model for sequence classification with three output labels. These pre-trained components facilitate efficient natural language processing tasks, with the model specifically configured for multi-class classification tasks with three possible labels. Additionally, we set the model to use a GPU ('cuda') if available, otherwise, it uses the CPU .



```python
#initialize the model
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=3)

#resources configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)

# Freeze some layers
for param in model.bert.embeddings.parameters():
    param.requires_grad = False


#Fine-tune the BERT model
optimizer = optim.AdamW(model.parameters(), lr=2e-5)

num_epochs = 7

# Learning rate scheduling
num_warmup_steps = 0
num_training_steps = num_epochs * len(train_dataloader)
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=num_warmup_steps, num_training_steps=num_training_steps)
```

24. Initializations for the Bert model.

After the initialization we added a freezing method for the embedding layers, the reason behind that is to prevent the parameters of the Bert model from being updated during fine tuning, as the often contain rich pretrained contextual information.

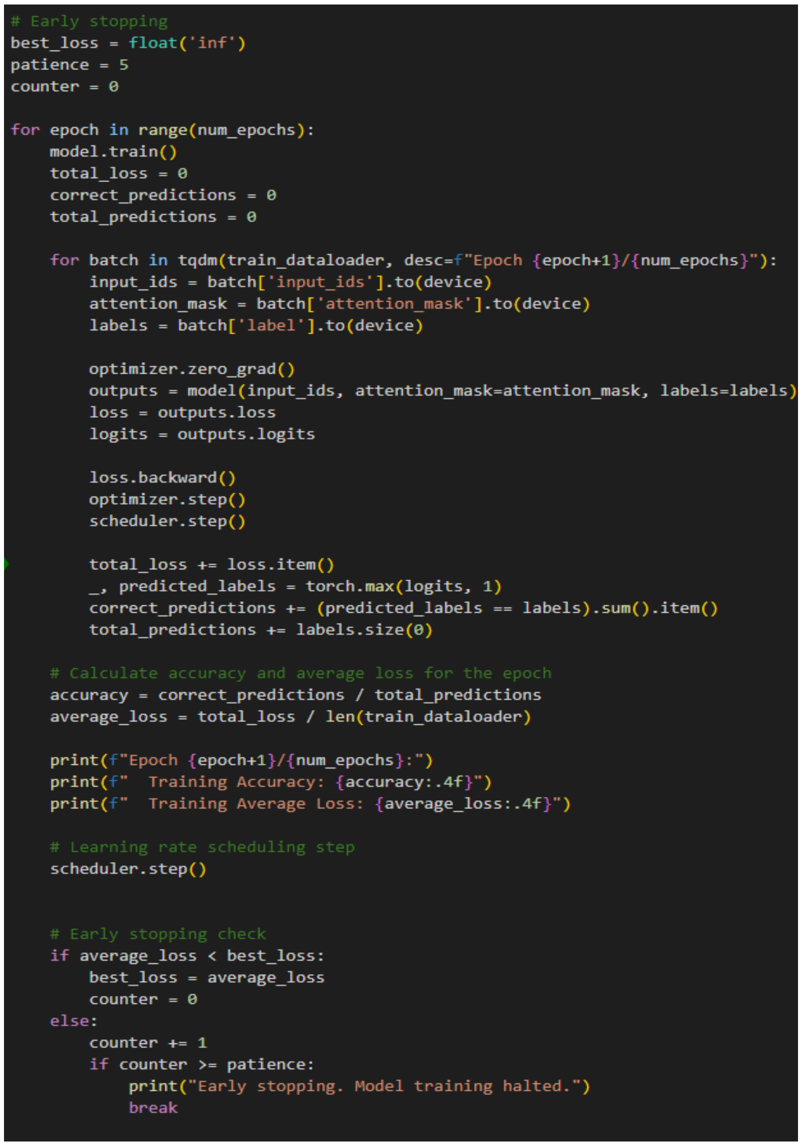
The next very important initialization is the implementation of the AdamW optimizer for updating the model parameters during training, with a learning rate of 2e-5. Specifically, AdamW is an optimization algorithm derived from Adam, designed to improve weight decay handling in neural network training. Introduced by Loshchilov and Hutter in 2017, AdamW decouples weight decay from the optimization step, mitigating its negative impact on adaptive learning rates. This modification enhances training stability and performance. For more details, you can refer to the original paper titled "Fixing Weight Decay Regularization in Adam" by I. Loshchilov and F. Hutter.

Furthermore, the number of epochs we are going to use are set to 7, that declares number of times the model will be trained on the entire training dataset. Also, we superinduce a learning rate scheduler which adjusts the learning rate during training to optimize convergence. Its purpose is to dynamically adapt the learning rate based on the model's performance, enhancing training stability, speeding up convergence in the initial phases, and ensuring accurate parameter updates as training progresses.

4.2 Bert model training

After the above initializations, now it is the time to implement them in the iteration of the training part. We begin by declaring three variables, the “best_loss” which Initializes the best loss to positive infinity for comparison during early stopping, the “patience” which determines the number of epochs to wait for accuracy improvement before stopping and the “counter” which is the one which calculates the unimproved iterated epochs.

First of all, we create a loop based on the number of declared epochs and we start by setting the model to training mode with "model.train() and our counters for training metrics. Then we create a batch loop which iterates through batches in the training data and displays a progress bar. Inside the loop we begin by transferring the data of each batch to the device.



```python
# Early stopping
best_loss = float('inf')
patience = 5
counter = 0

for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    correct_predictions = 0
    total_predictions = 0

    for batch in tqdm(train_dataloader, desc=f"Epoch {epoch+1}/{num_epochs}"):
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['label'].to(device)

        optimizer.zero_grad()
        outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        logits = outputs.logits

        loss.backward()
        optimizer.step()
        scheduler.step()

        total_loss += loss.item()
        _, predicted_labels = torch.max(logits, 1)
        correct_predictions += (predicted_labels == labels).sum().item()
        total_predictions += labels.size(0)

    # Calculate accuracy and average loss for the epoch
    accuracy = correct_predictions / total_predictions
    average_loss = total_loss / len(train_dataloader)

    print(f"Epoch {epoch+1}/{num_epochs}:")
    print(f"  Training Accuracy: {accuracy:.4f}")
    print(f"  Training Average Loss: {average_loss:.4f}")

    # Learning rate scheduling step
    scheduler.step()


    # Early stopping check
    if average_loss < best_loss:
        best_loss = average_loss
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print("Early stopping. Model training halted.")
            break
```

25. Building of the Bert model 1.1

- **batch['input_ids']:**

  Refers to a sequence of numerical tokens representing a batch of input text data. In natural language processing, these input IDs are often indices corresponding to words in a vocabulary. These tokenized input IDs serve as the model's input, enabling it to process and learn patterns within the text data.

- **batch['attention_mask']:**

  An attention mask in the context of neural networks, particularly in natural language processing tasks, is a binary matrix indicating which tokens in a sequence should be attended to and which should be ignored. It helps our model to focus on relevant parts of input sequences during processing. In the Transformer architecture, attention masks are used to prevent attending to padding tokens, ensuring that the model considers only actual tokens for information processing, allowing for more effective and efficient learning of contextual relationships in the input data.

- **batch['label'] :**

  Represents the ground truth labels associated with a batch of input data in supervised learning tasks. In our natural language processing, are numerical indices indicating the correct category for each corresponding input sample (0 , 1 or 3). During training, the model compares its predictions to these labels, and the discrepancy guides parameter updates through backpropagation. Accurate labels are essential for the model to learn and generalize effectively, enabling it to make meaningful predictions on unseen data.

- **optimizer.zero_grad():**

  Before computing the gradients for the model parameters, this line clears any previously accumulated gradients.

- **outputs = model(input_ids, attention_mask=attention_mask, labels=labels):**

  This line of code signifies a crucial step in the training process of a BERT-based model for sequence classification. This line conducts a forward pass through the neural network, where the model processes input sequences and generates predictions. Additionally, 'outputs' variable capture and enfold the results of the forward pass, including the computed loss and raw logits. These logits represent the unnormalized scores assigned to each class, serving as the basis for subsequent steps such as backpropagation and parameter updates during the training iteration. In a few words, it transforms input sequences into meaningful predictions while facilitating the optimization process through the computation of a training loss.

- **loss = outputs.loss:**

  Extracts the loss from the model's output. For sequence classification tasks, the model's output includes a loss attribute, representing the difference between the predicted logits and the true labels. This loss is a measure of how well the model is performing on the given batch.

- **logits = outputs.logits:**

  Retrieves the raw logits from the model's output. Logits are the unnormalized predictions before applying a softmax function. They represent the model's confidence scores for each class. These logits can later be used to compute probabilities or make predictions based on a chosen threshold.

- **loss.backward():**

  This line computes the gradients of the model's parameters with respect to the training loss. These gradients represent the direction and magnitude of the steepest ascent in the loss landscape. They quantify how much each parameter contributed to the error, providing crucial information for adjusting the model's weights.

- optimizer.step(): After computing the gradients, this line updates the model's parameters using the optimization algorithm (AdamW, in this case). The optimizer adjusts the weights to minimize the loss, employing the computed gradients and the learning rate. This step is essential for iteratively improving the model's performance.

- **scheduler.step():**

  Inside the loop it adjusts the learning rate for each optimization step. The learning rate scheduler (scheduler) adjusts the learning rate during training. Here, it's utilizing a linear schedule with warm-up. It ensures a gradual increase in the learning rate during the initial training steps, followed by a linear decay. Dynamic learning rate adjustments contribute to stable and efficient model convergence.

- **total_loss += loss.item():**

  This line accumulates the current batch's loss to the running total loss. It is a running sum that represents the cumulative training loss over all batches processed during the current epoch. This value is later used to calculate the average loss for the epoch.

- **_, predicted_labels = torch.max(logits, 1):**

  The torch.max(logits, 1) operation is used to predict the class labels for each input sequence in the batch. The model's raw logits, representing class scores, are passed through a softmax function internally, and torch.max returns the predicted class indices. The variable predicted_labels holds these predicted class labels.

- **correct_predictions += (predicted_labels == labels).sum().item():**

  This line calculates the number of correct predictions within the batch. It compares the predicted labels (predicted_labels) with the true labels (labels) and sums up the cases where they match. The resulting count is added to the running total of correct predictions.

- **total_predictions += labels.size(0):**

  The variable total_predictions keeps track of the total number of predictions made during the current epoch. It increments by the batch size (labels.size(0)) on each iteration, representing the cumulative count of predictions made across all batches in the epoch.

  After the batch loop we calculate and print the progress of our training in every epoch.

- **accuracy = correct_predictions / total_predictions:**

  This line calculates the accuracy for the current epoch. The correct_predictions variable holds the total number of correctly predicted instances, and total_predictions represents the total number of predictions made by the model during the epoch. Dividing the number of correct predictions by the total predictions yields accuracy, indicating the proportion of correctly classified instances.

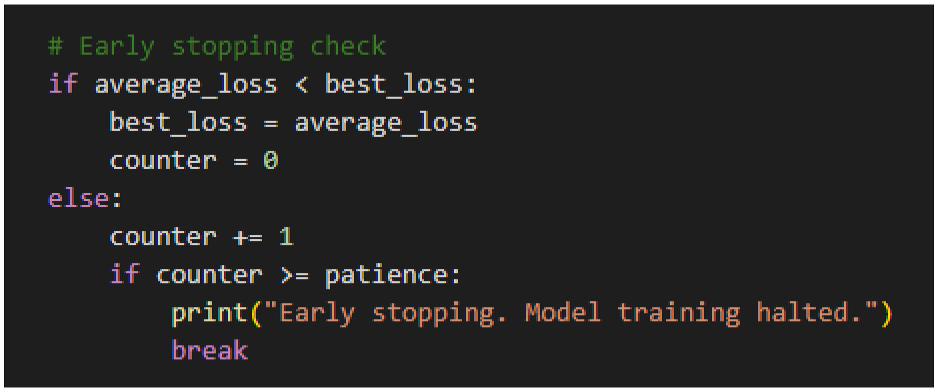
- **average_loss = total_loss / len(train_dataloader):**

  Computes the average training loss for the epoch. The total_loss variable accumulates the sum of losses across all batches during the epoch, and dividing this sum by the total number of batches (len(train_dataloader)) provides the average loss. This metric helps assess the overall performance of the model on the training data, providing insights into its ability to minimize the error across different batches.

- **scheduler.step():**

  after the loop allows for a final adjustment of the learning rate at the end of each epoch. This step ensures that the learning rate is appropriately updated for the next epoch.

At the end of each epoch iteration in this early stopping check we monitor the trend of the average training loss. If the loss does not improve for a certain number of consecutive epochs (determined by the patience parameter), the training loop is terminated to prevent overfitting and save computation resources.

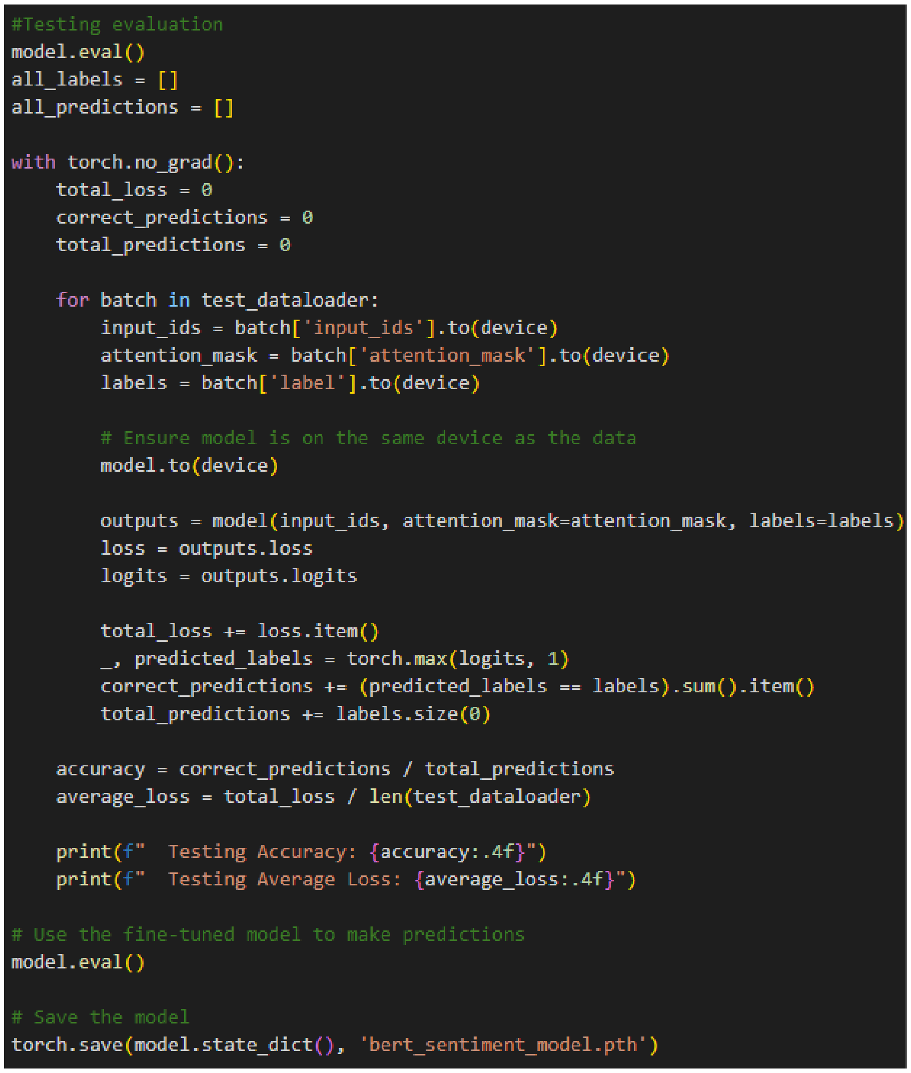


```python
# Early stopping check
if average_loss < best_loss:
    best_loss = average_loss
    counter = 0
else:
    counter += 1
    if counter >= patience:
        print("Early stopping. Model training halted.")
        break
```

26. Bert's training early stopping

4.3 Bert model testing evaluation

To begin with, in PyTorch, model.eval() is employed during evaluation to set the model in a state that deactivates training-specific operations like dropout and batch normalization. This ensures consistent inference results by maintaining fixed parameters. During evaluation, gradients are unnecessary, and these operations can introduce randomness.



```python
#Testing evaluation
model.eval()
all_labels = []
all_predictions = []

with torch.no_grad():
    total_loss = 0
    correct_predictions = 0
    total_predictions = 0

    for batch in test_dataloader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['label'].to(device)

        # Ensure model is on the same device as the data
        model.to(device)

        outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        logits = outputs.logits

        total_loss += loss.item()
        _, predicted_labels = torch.max(logits, 1)
        correct_predictions += (predicted_labels == labels).sum().item()
        total_predictions += labels.size(0)

    accuracy = correct_predictions / total_predictions
    average_loss = total_loss / len(test_dataloader)

    print(f"  Testing Accuracy: {accuracy:.4f}")
    print(f"  Testing Average Loss: {average_loss:.4f}")

# Use the fine-tuned model to make predictions
model.eval()

# Save the model
torch.save(model.state_dict(), 'bert_sentiment_model.pth')
```

27. Bert's training

By invoking model.eval(), the model behaves deterministically, guaranteeing reproducibility and preventing overfitting to the training set. This is essential for reliable assessments of a model's performance on unseen data, such as when calculating accuracy and loss during testing, as it avoids discrepancies introduced by training-specific mechanisms and contributes to robust model evaluation. Moreover, “all_labels “and “all_predictions” are Lists to store true labels and predicted labels for later use in the confusion matrix.

Next, we create our evaluation loop, generally this loop follows the training loop logic, so we are going to describe mostly it’s differences.

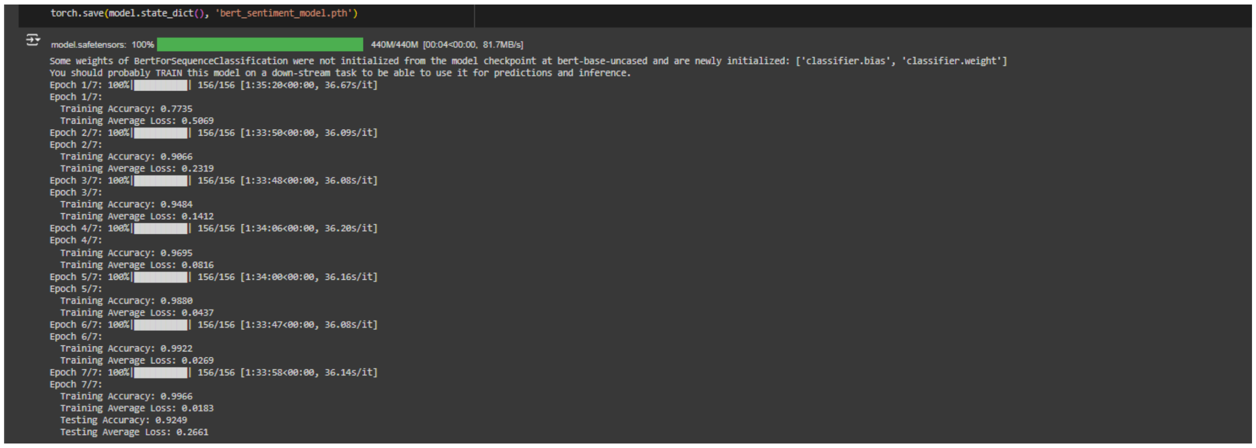
torch.no_grad(): This function initializes the loop which Inside this block, PyTorch disables gradient computation. This is done to save memory and processing power since during evaluation, you don't need to calculate gradients for backpropagation. It speeds up the inference process and reduces memory consumption. we initialize it with the “with torch.no_grad()”, inside this block, PyTorch disables gradient computation. This is done to save memory and processing power since during evaluation, you don't need to calculate gradients for backpropagation. It speeds up the inference process and reduces memory consumption.

for batch in test_dataloader: Just like the train batch loop, the test batch loop it Iterates through batches in the test dataloader. The test dataloader provides batches of data for evaluating the model on unseen examples. The only difference inside this loop is the “model.to(device)” line, which ensures the model and data are on the same device (CPU or GPU). This is necessary for proper computation, and it's especially important when using GPUs for acceleration.

Later on, we print the testing and average loss from the evaluation loop, we call **model.eval()** again to ensure that the model is in evaluation mode after the evaluation loop .

In summary, this evaluation loop is designed to rigorously assess the performance of the BERT model on the test data. It calculates accuracy and loss, crucial metrics for evaluating the model's effectiveness in making predictions on unseen examples. The model.eval() and torch.no_grad() ensure that the evaluation process is consistent, efficient, and does not interfere with the model's parameters or memory usage.

At the end we save the checkpoint of the epoch with the best accuracy in a dictionary which contains the entire state of the model, It includes all the learnable parameters and their current values.



```
torch.save(model.state_dict(), 'bert_sentiment_model.pth')

model.safetensors: 100% 440M/440M [00:04<00:00, 81.7MB/s]
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Epoch 1/7: 100%|██████████| 156/156 [1:35:20<00:00, 36.67s/it]
Epoch 1/7:
  Training Accuracy: 0.7735
  Training Average Loss: 0.5069
Epoch 2/7: 100%|██████████| 156/156 [1:33:50<00:00, 36.09s/it]
Epoch 2/7:
  Training Accuracy: 0.9066
  Training Average Loss: 0.2319
Epoch 3/7: 100%|██████████| 156/156 [1:33:48<00:00, 36.08s/it]
Epoch 3/7:
  Training Accuracy: 0.9484
  Training Average Loss: 0.1412
Epoch 4/7: 100%|██████████| 156/156 [1:34:06<00:00, 36.20s/it]
Epoch 4/7:
  Training Accuracy: 0.9695
  Training Average Loss: 0.0816
Epoch 5/7: 100%|██████████| 156/156 [1:34:00<00:00, 36.16s/it]
Epoch 5/7:
  Training Accuracy: 0.9880
  Training Average Loss: 0.0437
Epoch 6/7: 100%|██████████| 156/156 [1:33:47<00:00, 36.08s/it]
Epoch 6/7:
  Training Accuracy: 0.9922
  Training Average Loss: 0.0269
Epoch 7/7: 100%|██████████| 156/156 [1:33:58<00:00, 36.14s/it]
Epoch 7/7:
  Training Accuracy: 0.9966
  Training Average Loss: 0.0183
  Testing Accuracy: 0.9249
  Testing Average Loss: 0.2661
```

28. Bert’s results

Chapter 5

5.1 BiLSTM model creation.

After analyzing the Bert model, now it is time for the BiLSTM model. In natural language processing and text classification tasks, Bidirectional Long Short-Term Memory (BiLSTM) networks have proven to be highly effective. By incorporating information from both past and future states of a sequence, BiLSTMs excel at capturing contextual dependencies in text data. To create a BiLSTM model, we leverage libraries such as TensorFlow and Keras. Using the Tokenizer and pad_sequences functions from TensorFlow, we preprocess the text data, converting it into sequences of numerical tokens and ensuring uniform length.

With the Sequential model from Keras, we define our neural network architecture layer by layer, including an Embedding layer to generate word embeddings, Bidirectional LSTM layers for sequence processing, and a Dense layer for classification. Finally, we compile the model with an optimizer like Adam and train it on labeled data, achieving high accuracy in classifying text into predefined categories.

5.2 BiLSTM model initializations.

More specifically, we start by importing the libraries that we are going to use:

- Sequential: Sequential model for linear stack of layers, used for building deep learning models layer by layer.

- load_model: Loads a saved Keras model from a file for further use or evaluation.

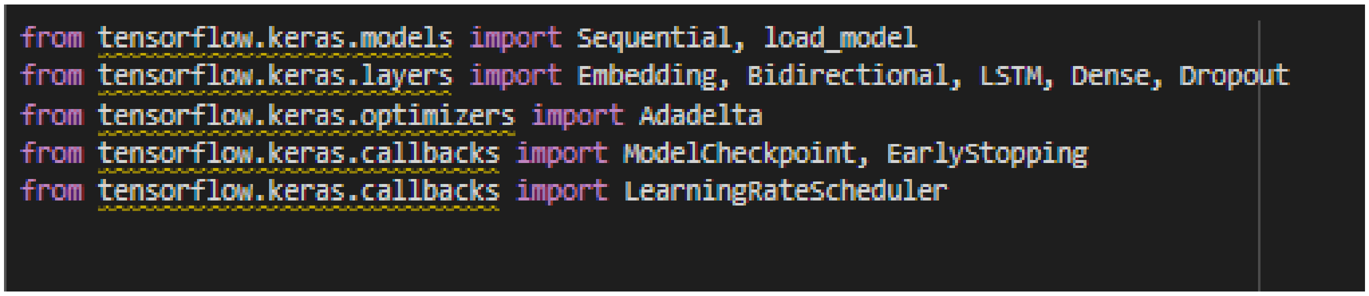
- Embedding: Converts integer indices to dense vectors of fixed size, representing word embeddings.

- Bidirectional: Wrapper for creating bidirectional recurrent neural networks, which process input sequences in both forward and backward directions.

- LSTM: Long Short-Term Memory layer, a type of recurrent neural network layer capable of learning long-term dependencies in sequential data.
- Dense: Fully connected layer where each neuron is connected to every neuron in the previous and next layers. evaluation and validation.
- ModelCheckpoint: Saves the model weights during training at specified checkpoints.

- EarlyStopping: Stops training when a monitored metric stops improving.

- LearningRateScheduler: Adjusts learning rate dynamically during training.



```
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Embedding, Bidirectional, LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adadelta
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.callbacks import LearningRateScheduler
```

29. BiLstm Libraries

5.3 AdaDelta Optimizer.

AdaDelta is a popular adaptive learning rate optimization algorithm designed for training neural networks. It was introduced by Matthew D. Zeiler in 2012 with a paper name << ADADELTA: AN ADAPTIVE LEARNING RATE METHOD >> , as an extension of the Adagrad algorithm. AdadDelta addresses some of the limitations of Adagrad, particularly its monotonically decreasing learning rate.

Unlike traditional optimization algorithms that use a fixed learning rate for all parameters, AdadDelta adapts the learning rate for each parameter in the model based on the magnitude of recent gradients. This adaptivity allows AdadDelta to perform well across a wide range of tasks and architectures without requiring manual tuning of the learning rate.

One of the key features of AdadDelta is its ability to maintain a moving average of parameter updates and gradients, which are used to compute the learning rate adjustments. By incorporating this historical information, AdadDelta can adaptively adjust the learning rates to different scales of parameters and gradients, making it robust to noisy gradients and sparse data.

AdadDelta also introduces the concept of an "adapting window" or "running average" of gradients and parameter updates. This window controls the memory of past gradients and updates, allowing AdadDelta to adaptively adjust the learning rates over time and handle non-stationary optimization problems effectively.

Another important aspect of AdadDelta is its computational efficiency. Unlike Adagrad, which requires storing the entire history of gradients, AdadDelta only stores a fixed-size window of past gradients and updates. This makes AdadDelta more memory-efficient and suitable for training large-scale neural networks.

Moreover, AdadDelta addresses the issue of the monotonically decreasing learning rate in Adagrad by using a moving average of squared parameter updates in the denominator of the learning rate computation. This helps stabilize the learning process and prevents the learning rate from becoming too small, which can hinder convergence.

Overall, AdadDelta is a powerful optimization algorithm for training neural networks, offering adaptive learning rates, robustness to noisy gradients, computational efficiency, and stability during training. Its simplicity and effectiveness make it a popular choice.

5.4 The key differences between AdaDelta and AdamW.

1. Update Rule:

AdaDelta: uses the root mean square (RMS) of parameter updates scaled by the root mean square of recent gradients to adapt the learning rate.

AdamW: (Weight Decay with Adam) is a variant of Adam that incorporates weight decay directly into the update step, making it compatible with modern weight decay techniques.

2. Parameter Update:

AdaDelta: uses only the gradient information to update the parameters.

AdamW: incorporates both gradient information and weight decay directly into the parameter update step.

3. Adaptivity:

AdaDelta: adapts the learning rate independently for each parameter based on the history of gradients for that parameter.

AdamW: uses adaptive learning rates for each parameter, but also incorporates the effect of weight decay regularization into the parameter update, which can improve generalization.

4. Learning Rate Adjustment:

AdaDelta: dynamically adjusts the learning rates for each parameter based on the moving average of previous gradients and updates.

AdamW: adjusts the learning rates for each parameter based on the estimation of first and second moments of gradients, but it also incorporates weight decay directly into the update, which helps in stabilizing the training process.

5. Implementation:

AdaDelta: typically requires fewer hyperparameters to tune compared to AdamW.

AdamW: requires tuning of hyperparameters such as learning rate, beta1, beta2, and weight decay parameter.

6. Memory Usage:

AdaDelta: maintains a history of gradients and parameter updates for each parameter, which can consume memory.

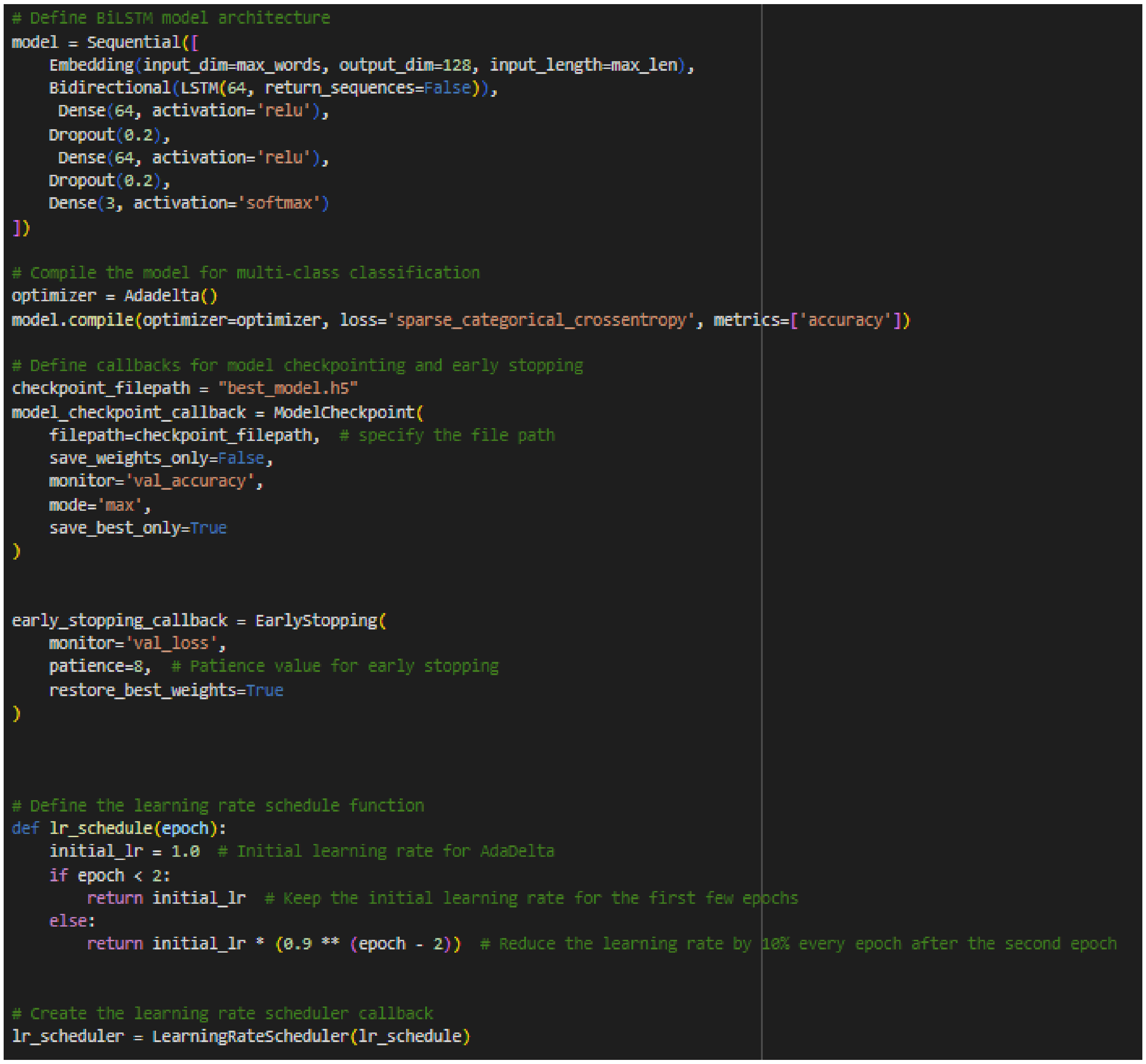
AdamW: also maintains a history of gradients and parameter updates, but the incorporation of weight decay might influence memory usage differently compared to AdaDelta.

In summary, while both AdaDelta and AdamW are adaptive optimization algorithms for training neural networks, AdamW is specifically designed to incorporate weight decay regularization directly into the update step, potentially leading to better generalization and stability during training, especially in the presence of large learning rates.

However, the choice between them often depends on factors like the specific problem, available computational resources, and empirical performance on the task at hand.

5.5 BiLstm architecture.

In the case of the creation of BiLstm model, it includes buidling the architecture of it, by defining the density of the neural network to 3 dense layers of 64 density, until the final layer of the 3 outputs. Later we set the optimizer that we are going to use followed by the compiling of the model for our multi-class classification. Let’s take a deeper dive by explaining the preparation of the training process in more detail.



```
# Define BiLSTM model architecture
model = Sequential([
    Embedding(input_dim=max_words, output_dim=128, input_length=max_len),
    Bidirectional(LSTM(64, return_sequences=False)),
     Dense(64, activation='relu'),
    Dropout(0.2),
     Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(3, activation='softmax')
])

# Compile the model for multi-class classification
optimizer = Adadelta()
model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Define callbacks for model checkpointing and early stopping
checkpoint_filepath = "best_model.h5"
model_checkpoint_callback = ModelCheckpoint(
    filepath=checkpoint_filepath,  # specify the file path
    save_weights_only=False,
    monitor='val_accuracy',
    mode='max',
    save_best_only=True
)


early_stopping_callback = EarlyStopping(
    monitor='val_loss',
    patience=8,  # Patience value for early stopping
    restore_best_weights=True
)



# Define the learning rate schedule function
def lr_schedule(epoch):
    initial_lr = 1.0  # Initial learning rate for AdaDelta
    if epoch < 2:
        return initial_lr  # Keep the initial learning rate for the first few epochs
    else:
        return initial_lr * (0.9 ** (epoch - 2))  # Reduce the learning rate by 10% every epoch after the second epoch


# Create the learning rate scheduler callback
lr_scheduler = LearningRateScheduler(lr_schedule)
```

30. BiLstm Architecture

1. model = Sequential

This line initializes a Sequential model, which allows for the linear stacking of layers. The Sequential model is simple and straightforward, ideal for building plain stacks of layers where each layer has exactly one input tensor and one output tensor.

2. **Embedding(input_dim=max_words,output_dim=128, input_length=max_len),**

The Embedding layer turns positive integers (indexes) into dense vectors of fixed size. It is commonly used as the first layer in a model to handle text data by converting words into vectors.

3. **Bidirectional(LSTM(64, return_sequences=False)),**

This wrapper allows the LSTM layer to process the input sequence in both forward and backward directions, capturing dependencies from both past and future contexts. LSTM(64): This specifies a Long Short-Term Memory (LSTM) layer with 64 units (neurons).

When return_sequences=False, the LSTM layer will return only the output of the last time step. This means that the layer outputs a single vector for each input sequence, which is typically the final hidden state of the LSTM after processing the entire sequence. This setting is used when the subsequent layers (such as dense layers) require a fixed-size input, rather than a sequence of outputs from each time step.

4. **Dense(64, activation='relu'),**

This Dense layer has 64 units and uses the ReLU activation function. The ReLU activation introduces non-linearity, which helps the network to learn complex patterns by adding the capacity to the model.

5. Dropout(0.2),

Dropout is a regularization technique where a fraction (0.2 or 20% in this case) of the input units are randomly set to 0 at each update during training time, which helps prevent overfitting.

6. Dense(3, activation='softmax')

The final Dense layer with 3 units and softmax activation function. The softmax function outputs a probability distribution over 3 classes, making it suitable for multi-class classification tasks.

7. optimizer = Adadelta()

As we have already said on the previous detailed of the AdaDelta optimizer. AdaDelta is an optimization algorithm that adapts learning rates based on a moving window of gradient updates, making it robust and well-suited for problems with sparse gradients or noisy data, making it most suitable on our case in which we have a small amount of data.

8. model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

This line compiles the model with the Adadelta optimizer, using sparse categorical crossentropy as the loss function, which is appropriate for integer-labeled multi-class classification tasks. The accuracy metric is used to evaluate the model's performance.

9. **model_checkpoint_callback = ModelCheckpoint(**

ModelCheckpoint is a callback function that saves the model after every epoch. This callback is useful to ensure that the best performing model on the validation set is saved during training.

10. **early_stopping_callback = EarlyStopping(**

Also an EarlyStopping function is used so the training it can be stopped automatically when a monitored metric ('val_loss') has stopped improving. This is useful to prevent overfitting and to save training time by stopping early if the model performance is getting worse, in our training if the performance is not improved after 8 epochs the training stops .

11. **def lr_schedule(epoch):**

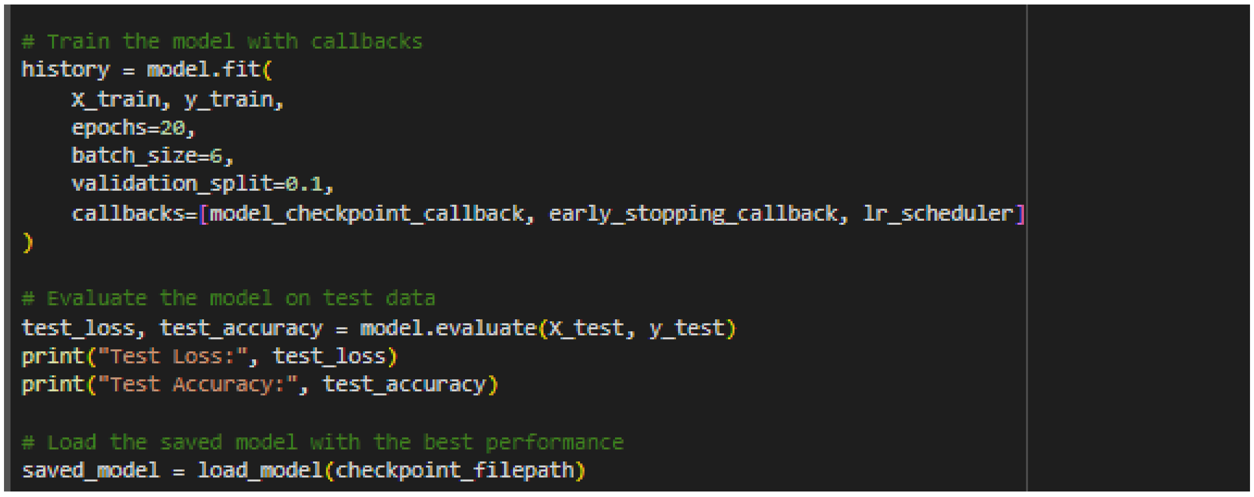
Defines a function to adjust the learning rate based on the epoch number. This custom learning rate scheduler helps improve our model convergence by reducing the learning rate as training progresses. In our learning rate scheduler we define an initial leaning rate to 1.0 and after training on 2 epochs we starting reducing the leaning rate by 0.1 in every epoch.

12. lr_scheduler = LearningRateScheduler(lr_schedule):

Initializes the LearningRateScheduler callback with the defined learning rate schedule function. This callback dynamically adjusts the learning rate according to the schedule during training.

5.6 BiLstm training.

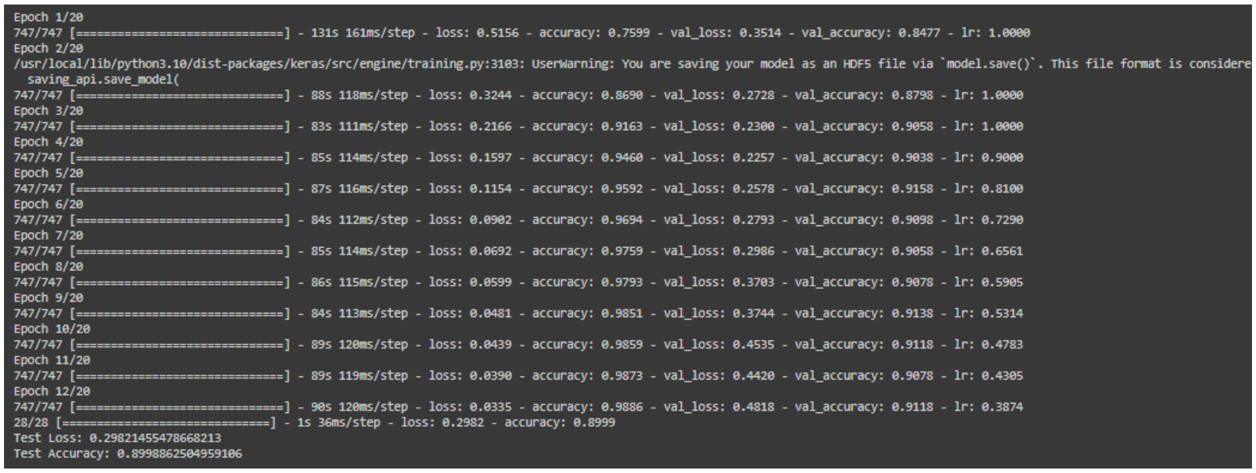
For the training procedure, we set the number of epochs to be 20 and our batch size to be 6, a small but a good option for our small dataset. In addition, we set the validation split to 0.1, that means that an amount of 10% is going to be kept from the training data for validation purposes. Furthermore, at the end of the model.fit function, we also call our callbacks which we mentioned previously.



```
# Train the model with callbacks
history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=6,
    validation_split=0.1,
    callbacks=[model_checkpoint_callback, early_stopping_callback, lr_scheduler]
)

# Evaluate the model on test data
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)

# Load the saved model with the best performance
saved_model = load_model(checkpoint_filepath)
```

31. BiLstm Training

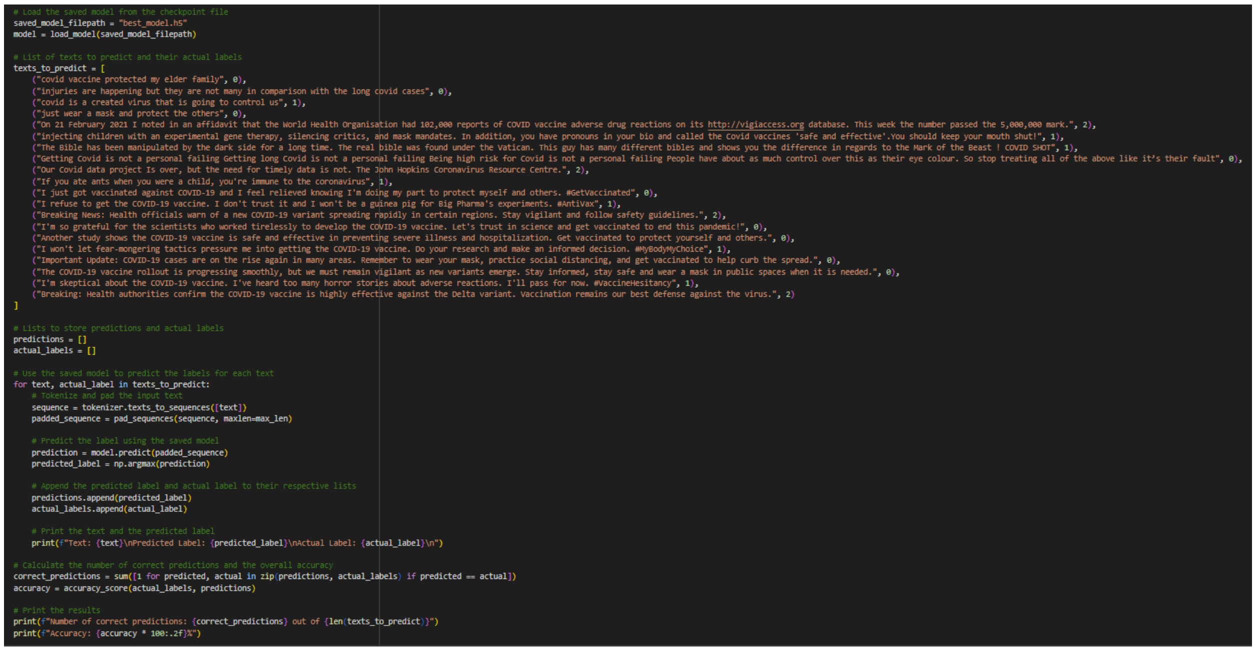
After the training we evaluate our model on the test data and then we save the best training step of our training procedure. The results of the training can be seen below.



```
Epoch 1/20
747/747 [==============================] - 131s 161ms/step - loss: 0.5156 - accuracy: 0.7599 - val_loss: 0.3514 - val_accuracy: 0.8477 - lr: 1.0000
Epoch 2/20
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This file format is considere
  saving_api.save_model(
747/747 [==============================] - 88s 118ms/step - loss: 0.3244 - accuracy: 0.8690 - val_loss: 0.2728 - val_accuracy: 0.8798 - lr: 1.0000
Epoch 3/20
747/747 [==============================] - 83s 111ms/step - loss: 0.2166 - accuracy: 0.9163 - val_loss: 0.2300 - val_accuracy: 0.9058 - lr: 1.0000
Epoch 4/20
747/747 [==============================] - 85s 114ms/step - loss: 0.1597 - accuracy: 0.9460 - val_loss: 0.2257 - val_accuracy: 0.9038 - lr: 0.9000
Epoch 5/20
747/747 [==============================] - 87s 116ms/step - loss: 0.1154 - accuracy: 0.9592 - val_loss: 0.2578 - val_accuracy: 0.9158 - lr: 0.8100
Epoch 6/20
747/747 [==============================] - 84s 112ms/step - loss: 0.0902 - accuracy: 0.9694 - val_loss: 0.2793 - val_accuracy: 0.9098 - lr: 0.7290
Epoch 7/20
747/747 [==============================] - 85s 114ms/step - loss: 0.0692 - accuracy: 0.9759 - val_loss: 0.2986 - val_accuracy: 0.9058 - lr: 0.6561
Epoch 8/20
747/747 [==============================] - 86s 115ms/step - loss: 0.0599 - accuracy: 0.9793 - val_loss: 0.3703 - val_accuracy: 0.9078 - lr: 0.5905
Epoch 9/20
747/747 [==============================] - 84s 113ms/step - loss: 0.0481 - accuracy: 0.9851 - val_loss: 0.3744 - val_accuracy: 0.9138 - lr: 0.5314
Epoch 10/20
747/747 [==============================] - 89s 120ms/step - loss: 0.0439 - accuracy: 0.9859 - val_loss: 0.4535 - val_accuracy: 0.9118 - lr: 0.4783
Epoch 11/20
747/747 [==============================] - 89s 119ms/step - loss: 0.0390 - accuracy: 0.9873 - val_loss: 0.4420 - val_accuracy: 0.9078 - lr: 0.4305
Epoch 12/20
747/747 [==============================] - 90s 120ms/step - loss: 0.0335 - accuracy: 0.9886 - val_loss: 0.4818 - val_accuracy: 0.9118 - lr: 0.3874
28/28 [==============================] - 1s 36ms/step - loss: 0.2982 - accuracy: 0.8999
Test Loss: 0.29821455478668213
Test Accuracy: 0.8998862504959106
```

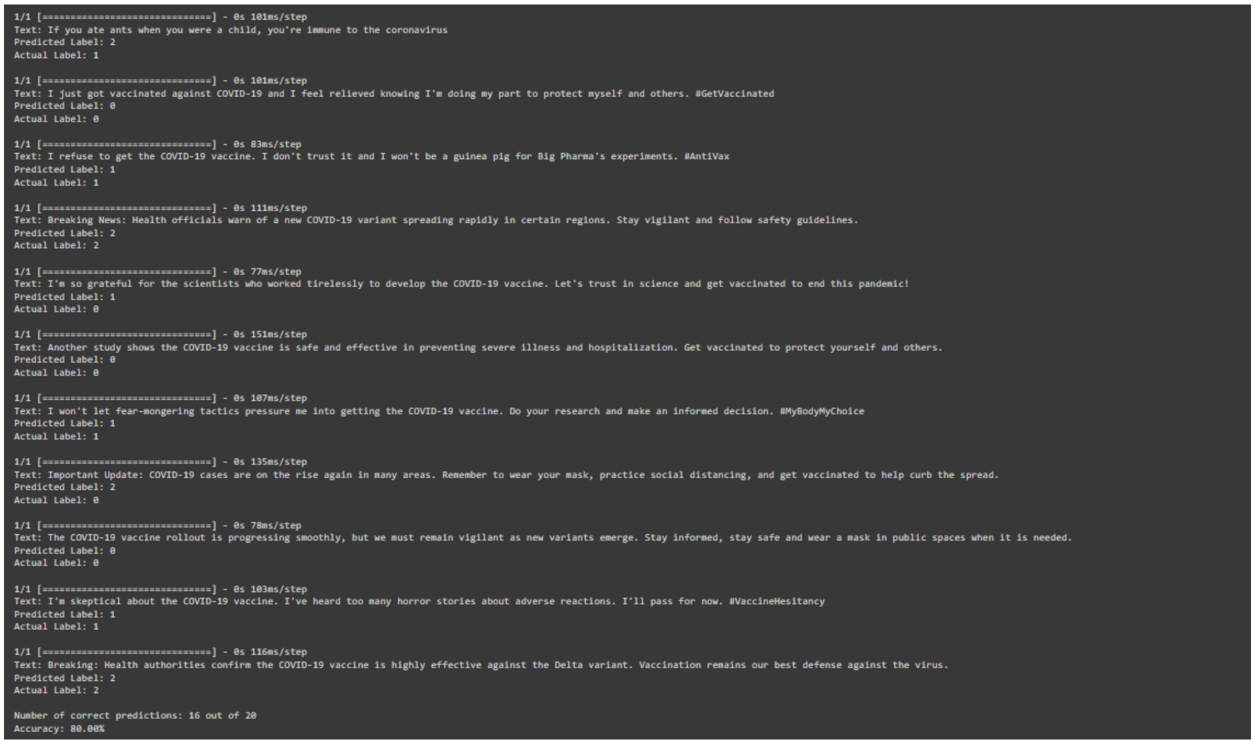
32. BiLstm Training Results

We also used some random data that are not included in our dataset to check out the predictions of our model:



```python
# Load the saved model from the checkpoint file
saved_model_filepath = "best_model.h5"
model = load_model(saved_model_filepath)

# List of texts to predict and their actual labels
texts_to_predict = [
    ("covid vaccine protected my elder family", 0),
    ("injuries are happening but they are not many in comparison with the long covid cases", 0),
    ("covid is a created virus that is going to control us", 1),
    ("just wear a mask and protect the others", 0),
    ("On 21 February 2021 I noted in an affidavit that the World Health Organisation had 102,000 reports of COVID vaccine adverse drug reactions on its http://vigiaccess.org database. This week the number passed the 5,000,000 mark.", 2),
    ("injecting children with an experimental gene therapy, silencing critics, and mask mandates. In addition, you have pronouns in your bio and called the Covid vaccines 'safe and effective'.You should keep your mouth shut!", 1),
    ("The Bible has been manipulated by the dark side for a long time. The real bible was found under the Vatican. This guy has many different bibles and shows you the difference in regards to the Mark of the Beast ! COVID SHOT", 1),
    ("Getting Covid is not a personal failing Getting long Covid is not a personal failing Being high risk for Covid is not a personal failing People have about as much control over this as their eye colour. So stop treating all of the above like it’s their fault", 0),
    ("Our Covid data project Is over, but the need for timely data is not. The John Hopkins Coronavirus Resource Centre.", 2),
    ("If you ate ants when you were a child, you're immune to the coronavirus", 1),
    ("I just got vaccinated against COVID-19 and I feel relieved knowing I'm doing my part to protect myself and others. #GetVaccinated", 0),
    ("I refuse to get the COVID-19 vaccine. I don't trust it and I won't be a guinea pig for Big Pharma's experiments. #AntiVax", 1),
    ("Breaking News: Health officials warn of a new COVID-19 variant spreading rapidly in certain regions. Stay vigilant and follow safety guidelines.", 2),
    ("I'm so grateful for the scientists who worked tirelessly to develop the COVID-19 vaccine. Let's trust in science and get vaccinated to end this pandemic!", 0),
    ("Another study shows the COVID-19 vaccine is safe and effective in preventing severe illness and hospitalization. Get vaccinated to protect yourself and others.", 0),
    ("I won't let fear-mongering tactics pressure me into getting the COVID-19 vaccine. Do your research and make an informed decision. #MyBodyMyChoice", 1),
    ("Important Update: COVID-19 cases are on the rise again in many areas. Remember to wear your mask, practice social distancing, and get vaccinated to help curb the spread.", 0),
    ("The COVID-19 vaccine rollout is progressing smoothly, but we must remain vigilant as new variants emerge. Stay informed, stay safe and wear a mask in public spaces when it is needed.", 0),
    ("I'm skeptical about the COVID-19 vaccine. I've heard too many horror stories about adverse reactions. I'll pass for now. #VaccineHesitancy", 1),
    ("Breaking: Health authorities confirm the COVID-19 vaccine is highly effective against the Delta variant. Vaccination remains our best defense against the virus.", 2)
]

# Lists to store predictions and actual labels
predictions = []
actual_labels = []

# Use the saved model to predict the labels for each text
for text, actual_label in texts_to_predict:
    # Tokenize and pad the input text
    sequence = tokenizer.texts_to_sequences([text])
    padded_sequence = pad_sequences(sequence, maxlen=max_len)

    # Predict the label using the saved model
    prediction = model.predict(padded_sequence)
    predicted_label = np.argmax(prediction)

    # Append the predicted label and actual label to their respective lists
    predictions.append(predicted_label)
    actual_labels.append(actual_label)

    # Print the text and the predicted label
    print(f"Text: {text}\nPredicted Label: {predicted_label}\nActual Label: {actual_label}\n")

# Calculate the number of correct predictions and the overall accuracy
correct_predictions = sum([1 for predicted, actual in zip(predictions, actual_labels) if predicted == actual])
accuracy = accuracy_score(actual_labels, predictions)

# Print the results
print(f"Number of correct predictions: {correct_predictions} out of {len(texts_to_predict)}")
print(f"Accuracy: {accuracy * 100:.2f}%")
```

33. Creation of dummy data



```
1/1 [==============================] - 0s 101ms/step
Text: If you ate ants when you were a child, you're immune to the coronavirus
Predicted Label: 2
Actual Label: 1

1/1 [==============================] - 0s 101ms/step
Text: I just got vaccinated against COVID-19 and I feel relieved knowing I'm doing my part to protect myself and others. #GetVaccinated
Predicted Label: 0
Actual Label: 0

1/1 [==============================] - 0s 83ms/step
Text: I refuse to get the COVID-19 vaccine. I don't trust it and I won't be a guinea pig for Big Pharma's experiments. #AntiVax
Predicted Label: 1
Actual Label: 1

1/1 [==============================] - 0s 111ms/step
Text: Breaking News: Health officials warn of a new COVID-19 variant spreading rapidly in certain regions. Stay vigilant and follow safety guidelines.
Predicted Label: 2
Actual Label: 2

1/1 [==============================] - 0s 77ms/step
Text: I'm so grateful for the scientists who worked tirelessly to develop the COVID-19 vaccine. Let's trust in science and get vaccinated to end this pandemic!
Predicted Label: 1
Actual Label: 0

1/1 [==============================] - 0s 151ms/step
Text: Another study shows the COVID-19 vaccine is safe and effective in preventing severe illness and hospitalization. Get vaccinated to protect yourself and others.
Predicted Label: 0
Actual Label: 0

1/1 [==============================] - 0s 107ms/step
Text: I won't let fear-mongering tactics pressure me into getting the COVID-19 vaccine. Do your research and make an informed decision. #MyBodyMyChoice
Predicted Label: 1
Actual Label: 1

1/1 [==============================] - 0s 135ms/step
Text: Important Update: COVID-19 cases are on the rise again in many areas. Remember to wear your mask, practice social distancing, and get vaccinated to help curb the spread.
Predicted Label: 2
Actual Label: 0

1/1 [==============================] - 0s 78ms/step
Text: The COVID-19 vaccine rollout is progressing smoothly, but we must remain vigilant as new variants emerge. Stay informed, stay safe and wear a mask in public spaces when it is needed.
Predicted Label: 0
Actual Label: 0

1/1 [==============================] - 0s 103ms/step
Text: I'm skeptical about the COVID-19 vaccine. I've heard too many horror stories about adverse reactions. I'll pass for now. #VaccineHesitancy
Predicted Label: 1
Actual Label: 1

1/1 [==============================] - 0s 116ms/step
Text: Breaking: Health authorities confirm the COVID-19 vaccine is highly effective against the Delta variant. Vaccination remains our best defense against the virus.
Predicted Label: 2
Actual Label: 2

Number of correct predictions: 16 out of 20
Accuracy: 80.00%
```

34. Prediction results on the dummy data

Chapter 6

6.1 Conclusions.

This thesis covered in detail the topic of text classification using a BERT and a BiLSTM model to classify tweets related to COVID-19. First, we provided a general description of the technologies and techniques used in neural network models. Next, we detailed the code used, starting with data preparation for the two models and outlining the differences and similarities between BERT and BiLSTM.

We also created graphs to represent the data statistics, allowing us to observe the state and content of the data. Subsequently, we analyzed the libraries, optimizers, and other techniques employed. Additionally, almost every line of the models was explained, including the rationale for their use, and compared to the other model.

6.2 Improvements and Future work.

First of all, it is widely known that COVID-19 was a very delicate situation, and so were the opinions that everyone developed. Alongside this, scientific discoveries and laws kept changing and still do, making it difficult in some cases to classify an opinion with just a number from 0 to 1.

With this in mind, the specific dataset can be improved and maintained in the future. Also, if the Twitter (X) API remains accessible and free, it will be extremely beneficial for the improvement of the dataset by adding more training data, making it more precise and covering more COVID-19 topics. With that said, some changes and tuning could be done again if the dataset is changed.

6.3 Bibliography.

1. https://huggingface.co/tasks/text-classification
2. https://builtin.com/data-science/recurrent-neural-networks-and-lstm
3. https://thinkingneuron.com/sentiment-analysis-of-tweets-using-bert/#DistilBERT
4. http://arxiv.org/pdf/1711.05101v2/1000
5. https://github.com/AminHasibul/ConspiracyAgaintstCovidVaccines/tree/main
6. https://www.altexsoft.com/blog/preparing-your-dataset-for-machine-learning-8-basic-techniques-that-make-your-data-better/
7. https://www.analyticsvidhya.com/blog/2021/12/fine-tune-bert-model-for-sentiment-analysis-in-google-colab/
8. ADADELTA: AN ADAPTIVE LEARNING RATE METHOD
9. Attention Is All You Need
10. https://towardsdatascience.com/sentence-classification-using-bi-lstm-b74151ffa565
11. https://www.researchgate.net/figure/Results-of-five-optimizers-applied-to-Bi-LSTM-CRF_tbl3_344375190
12. https://www.researchgate.net/publication/328333982_A_Deep_Recurrent_Neural_Network_with_BiLSTM_model_for_Sentiment_Classification
13. https://deepdatascience.wordpress.com/2016/11/18/which-lstm-optimizer-to-use/
14. https://towardsdatascience.com/6-ways-to-improve-your-ml-model-accuracy-ec5c9599c436
15. https://towardsdatascience.com/how-to-decide-on-learning-rate-6b6996510c98
16. https://www.sciencedirect.com/science/article/pii/S0957417422017353
17. https://www.nature.com/articles/s41598-022-21604-7
18. https://content.iospress.com/articles/intelligent-decision-technologies/idt210058
19. https://onlinelibrary.wiley.com/doi/full/10.1155/2021/4321131
20. https://ojs.bonviewpress.com/index.php/JCCE/article/view/838
21. https://arxiv.org/abs/2103.11943
22. https://www.nature.com/articles/s41598-022-21604-7

23. https://dl.acm.org/doi/abs/10.1145/3409334.3452074
24. https://onlinelibrary.wiley.com/doi/full/10.1155/2022/3498123
25. https://books.google.gr/books?hl=el&lr=&id=uAPuDwAAQBAJ&oi=fnd&pg=PP1&dq=data+preparation+for+machine+learning&ots=Cm2Kzg9NqR&sig=dk6XMEKU7kuCXORBKqrmATzNtKw&redir_esc=y#v=onepage&q=data%20preparation%20for%20machine%20learning&f=false