6CCS3PRJ Final Year Project

Enhancing Fake News Detection With Unbiased Dataset, Explainability and BERT-based Models

Final Project Report

Author: Jan Marczak

Supervisor: Odinaldo Rodrigues

Student ID: 19029774

April 7, 2022

Abstract

Fake news and misinformation are becoming increasingly difficult to deal with. They present dangerous implications for society, particularly given the growing popularity of social media networks. A lot has been done in this research area to implement the best-performing machine learning models. However, there is a clear lack of thorough approaches that combine state-of-the-art NLP models with unbiased datasets, AI Explainability techniques and accessible software.

This project aims to provide such comprehensive solution in automatic misinformation detection. It analyses existing fake news datasets and eliminates their biased and flawed characteristics in order to create an improved dataset. This data is later utilised in training and experimenting with different BERT-based language models. The best suited and performing classifier is included in the website that besides predicting the veracity of a claim, provides its explanation.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Jan Marczak

April 7, 2022

Acknowledgements

I wish to thank my supervisor, Dr Odinaldo Rodrigues, for offering his support, guidance and expertise throughout this project. His constant help and availability have been invaluable in completing this project.

Contents

Chapter 1

Introduction

In recent years, with the rapid growth of social media, misinformation has been spreading at dangerous rates. According to a Pew Research Center Survey [1] conducted in 2021 almost half (48%) of adults in the United States get their news from social media. By transitioning from traditional news-consumption platforms like television or newspapers, social media users can not only read the news from questionable sources but also share them in a matter of a few clicks.

Lies spread by misinformation create a false reality and present a real threat to social stability and democracy. An example of such is a Facebook misinformation scandal during the 2016 presidential elections in the United States, where specific user demographics were targeted with propaganda-style fake information. More recently, the coronavirus pandemic has also underlined how disinformation can represent a risk to personal and public health. According to World Health Organization, [2], because of fake news, over 6000 people were hospitalized during the beginning of the COVID-19 pandemic.

Many websites such as *Politifact*[1] or *Snopes*[2] have existed for a long time to help with this. They employ several professional journalists to label articles, social media posts, and speeches as truthful or not. However, the magnitude of social media platforms, the rate of content creation, and dissemination call for automatic approaches. While a lot has been done in this research area the main issues that stand out are the lack of an unbiased, versatile dataset, limited model explanations and difficult accessibility to fact-checking tools.

The ambition of this project is to attempt to solve the problem of misinformation online with a thorough approach. It aims to enhance fake news classification methods with the use of

[1] `https://Politifact.com`
[2] `https://Snopes.com`

a newly and wisely created dataset, which is utilised to train deep learning transformer models like BERT and RoBERTa. The models themselves are explainable to provide users with a better understanding of why a particular claim is considered to be true or false. Moreover, a simple but intuitive desktop application is implemented to allow users to fact-check information in a matter of a few clicks.

1.1 Motivation

This section explains in more detail the main motivation behind solving the project with this particular methodology.

1.1.1 Lack of Datasets

Current automatic methods usually combine techniques of Information Retrieval, Natural Language Processing (NLP), and Machine or Deep Learning. Most of these solutions require a large-scale, trustworthy, and labelled dataset, with enough contrasting examples to be absorbed by machine learning algorithms. This unfortunately, is still considered to be a bottleneck, as most of the publicly available datasets present several issues like automated non-trustworthy labelling, lack of diverse entries, or small data size. Many researchers have already attempted to fight the spread of fake news through machine learning models using these datasets. Their proposed solutions usually achieve very impressive accuracy scores. However, this might be due to the potential bias and undesired characteristics present in publicly available datasets. This project aims to examine this hypothesis and provide a potential solution.

1.1.2 Lack of Explainability

Twitter and Instagram have already implemented their methods of labelling posts as misleading. Twitter uses a warning system for their textual posts [3] and Instagram does the same approach for images [4]. Whereas both platforms identify misinformation in different ways their solutions present one important defect: there is no explainability given to the user on why the post has been labelled as deceitful. Such solutions may only discourage users to believe in the trustworthiness of these techniques and inaccurate labelling may lead to increased scepticism among userbase. In fact, people have been fooling with Twitter's COVID-19 fake news detection system, by putting 5G and coronavirus in the same sentence, which automatically triggers the labelling on such tweet [5]. Moreover, Instagram is inaccurately hiding images that are just

a form of digital art [6]. As artificial intelligence technologies are being integrated with most aspects of human lives the need for AI explainability will grow, and the fake news detection problem is one of its prime examples.

1.2 Aims & Objectives

Following stated limitations of current approaches, a big portion of this project is analysing existing fake news datasets. This is done by inspecting suspicious characteristics of particular datasets that might seem to help the model achieve higher results, but in reality, create a flawed classifier. To offset this potential problem this project constructs a new dataset, by combining existing resources thoughtfully and sensibly. Moreover, this project aims to utilise cutting-edge technology for Natural Language Processing tasks, such as Transformer-based language models. To give users the ability to absorb information and verify claims more thoughtfully, additional Explainable AI (XAI) techniques are implemented. This may not only give users a better idea of the reasons for the particular prediction but also give developers a chance to understand their deep learning models. A website combining all deliverables of this project is also implemented, to make fake news detectors more accessible.

1.3 Report Structure

This report begins with a background overview of topics relevant to the project. A review of existing works in the field of misinformation detection, as well as an analysis and explanation of publicly available datasets have been done to put the project into context. Popular methods of explaining Machine Learning models have also been referenced, to better understand its available techniques.

Chapter 3 lists the requirements to be followed during the implementation phase of the project. The design builds upon them and focuses on why particular methodologies were chosen to achieve this project's deliverables.

Chapter 5 describes the implementation techniques that were carried out and goes into more depth about the development process. Evaluation serves as a discussion about the outcome of the project, where each of its main components is honestly assessed.

The report closes with the conclusion of the whole work and the future improvements that can enhance this project.

Chapter 2

Background

This chapter serves as a review of existing literature and methodologies applied in this project. It starts by defining a concept of fake news and follows a review of recent work on automatic fake news detection. Both traditional and deep machine learning approaches have been analysed. It then continues with the overview of publicly available datasets that exist for this classification problem and concludes with the outline of Explainable AI techniques.

2.1 Defining Fake News

The term 'Fake News' has been widely popularised after the 2016 presidential elections in the United States. According to *The Independent* [7], Donald Trump has called journalists and news outlets "fake news" nearly 2000 times during his presidency, taking this term into the mainstream. Cambridge Dictionary defines fake news as "false stories that appear to be news, spread on the internet or using other media, usually created to influence political views or as a joke.". The term is also often used in the context of more specific, misleading categories like satire, fabrication or propaganda. Misinformation and disinformation are its close synonyms, where the first is usually described as false information spread regardless of intentions, whereas the second has a deceiving purpose. Regardless of correct definitions and their intentions all fake news, disinformation and misinformation have a misleading effect on people. For this project's purpose, the paper uses them alternately and treats them as synonyms.

2.2 Traditional Machine Learning Algorithms

Traditional supervised machine learning algorithms have become a standard for solving categorical classification or regression problems. Their small data needs, a little requirement for computing power, simplicity and widely-available resources make them an attractive and easy option for many people. These classifiers learn from a given labelled data, where the selection of the particular classifier and features (inputs) to be fed into that algorithm are chosen by programmers.

The issue of fake news classification has already been solved extensively using traditional supervised algorithms. For example, researchers from [8] modelled this problem as a binary classification problem and used a variety of machine learning algorithms to evaluate the results. Five different classifiers have been used: Naive Bayes, Decision Trees, Support Vector Machines (SVM), Random Forrest, and XGBoost, with SVM and XGBoost yielding the best overall accuracy. The dataset used for training purposes consisted of a large number of tweets and over 100 different features extracted from the actual text or the user who posted it. They considered five different groups of attributes: user-level features, tweet-level features, text features, topic features, and sentiment features. They observed a noticeable decline from 0.90 to 0.77 in F1 score after excluding user-level features from the algorithm's learning process, proving their importance in this classification problem.

In fact, researchers from [9] concentrate their work on analysing the importance and influence of particular attributes in fake news detection. As described themselves, they "conduct a highly exploratory investigation that produced hundreds of thousands of models from a large and diverse set of features.". They established training sets of features randomly, allowing for unbiased results. Their findings suggest that different models with high accuracies identify distinct fake newsgroups, which may explain the broad definition of fake news mentioned in Section 2.1. This also suggests the high complexity of this classification problem as often features interact with each other in many different and complex ways.

2.3 Neural Network Algorithms

This section describes a subset of machine learning called neural networks, which are also the heart of deep learning algorithms. It starts with the most basic neural network and finishes with two instances of deep learning neural networks. Additionally, some examples of their usage in NLP tasks are provided. These explanations serve as a ground to cover later topics.

2.3.1 Artificial Neural Networks (ANN)

Artificial neural networks (also called feed-forward neural networks) are a form of more advanced machine-learning algorithms that mimic the functioning of the human brain. Contrary to traditional machine learning, neural networks use data that requires little human intervention to function properly, as they can learn and make intelligent decisions on their own. ANN nodes have input and output edges with associated weights and an activation threshold as a non-linear function. The activation function decides whether the neuron's input to the network is important enough to activate this particular neuron. The right choice of an activation function can lead to better performance. ReLU function has been widely used in the field due to its efficiency when compared to more standard sigmoid and tanh functions. Its popular alternatives also include Leaky ReLU, SELU and GELU functions. All of the listed activation functions try to solve the problems of dying ReLU, vanishing gradients or exploding gradients. Their in-depth analysis is conducted by researchers in [10]. During each iteration of ANN's learning process, the network adjusts its weights according to a learning rule and the calculated error (difference between processed output and target output). Backpropagation is the key to how a neural network learns a particular task. It computes the gradient of the loss function for a single input-output example. This gradient is then used by an optimization algorithm to update the model weights and minimize the loss function. The gradient descent algorithm and its extensions [11] is a common choice to perform the optimization in a neural network. Depending on their variant, they use different amounts of data to adjust the weights. They maintain a single learning rate for all weight updates, which does not change during training. The Adam [12] is an extension to stochastic gradient descent that is widely used in deep learning applications. Unlike previously described methods, it computes individual learning rates for different parameters. As described by the authors, the method combines the advantages of AdaGrad [13], which works well with sparse gradients and RMSProp, which works well with stochastic objectives.

2.3.2 Convolutional Neural Networks (CNN)

According to [14] convolutional networks are a specialized kind of neural network for processing data that has a known grid-like topology. They are simply neural networks that use a convolution process in place of general matrix multiplication in at least one of their layers. CNN tends to show very high accuracy in image recognition problems, as they can capture spatial features of data. However, they are incapable of effectively interpreting temporal information

with sequential data.

2.3.3 Recurrent Neural Networks (RNN)

Recurrent neural networks are different as they can memorise information from previous inputs to influence the current input and output. This means that the output is dependent on the elements from prior sequences. Hence, RNN is usually utilised for sequential data (text, audio, video, etc.) and can be used in 3 different architectures:

- **Encoder Network** - Takes a sequence and outputs a single vector.
- **Decoder Network** - Takes a fixed size vector and outputs a sequence of any size
- **Encoder-Decoder Network** - Take a sequence and outputs a sequence.

RNN are slow to train as their sequential flow does not utilise parallelization. For this, transformer models explained in Section 2.5 are used.

2.3.4 Neural Networks for NLP tasks

All three described networks have been extensively utilised for various NLP problems. Despite CNN being used primarily for computer vision, its much faster learning is attractive in comparison to RNN. For example, researchers in [15] utilise two parallel CNNs to extract latent features from both textual and visual information to achieve an F1 score of 0.92 on the Kaggle dataset [16]. Albeit slower, RNN is often the preferred choice among these networks to handle sequential data due to their nature. However, [17] found both CNN and RNN to perform equally good on two fake news datasets, where recurrent networks outperformed convolutional by only one percentage point. The proposed research work [18] obtained 99 % in F1 score using plain feed-forward neural networks (ANN) on the same Kaggle dataset [16], showing the potential of simpler approaches. This has also been brought up in papers tackling other text classification problems. [19] Applied both ANN and RNN for automating intrusion detection and found both to perform equally well. All these instances show that the choice of a neural network for particular tasks is not obvious, and ideally, experimentation between them has to be conducted to pick the appropriate one.

2.4 Word-Embedding

Whereas approaches mentioned in Section 2.2 and 2.3.4 provide good accuracy scores, their emphasis on text, user, or network level features diminishes the importance of the actual text itself. For its representation, it is common to apply word-embedding techniques. As explained in the [20] "the goal of word embedding is mapping the words in unlabelled text data to continuously-valued space, to capture the internal semantic and syntactic information".

There exist many word embedding techniques. In [21] **TF-IDF** was used, which is a combination of Term Frequency (TF) and Inverse Document Frequency (IDF) metrics. TF-score measures how often a certain word occurs in the document, relative to all possible words. IDF-score however, diminishes the weight of terms that occur very frequently. TF-IDF is based on the Bag-of-Words (BoW) model, and its main issue is that any information about the order or structure of words in the document is discarded.

More sophisticated, **Word2vec** [22] uses a neural network model to learn word associations from a large corpus of text. Such model is trained to learn embeddings that predict either the probability of a surrounding word occurring given a centre word (SkipGram) or vice versa (CBoW). It is however context-free, as a single word representation is generated for each word in the vocabulary. It also cannot handle out-of-vocabulary words well.

To consider full context and unseen words, more advanced deep learning transformer models that produce word embeddings such as **BERT** 2.5.2 should be used.

2.5 Transformer Models

The transformer model is based entirely on attention mechanisms, dispensing with recurrence and convolutions entirely [23]. This section describes the transformer neural network model and its components that serve as a prerequisite to understanding how BERT architecture works.

2.5.1 Transformer Neural Network

Transformer neural network architecture [23] shown in Figure 2.1 was introduced in 2017. The network employs an encoder-decoder architecture, similarly to RNN. Its main difference is that the input sequence can be passed in parallel, allowing for much faster training times. This architecture was a breakthrough in language translation and allowed for better results with higher efficiency.

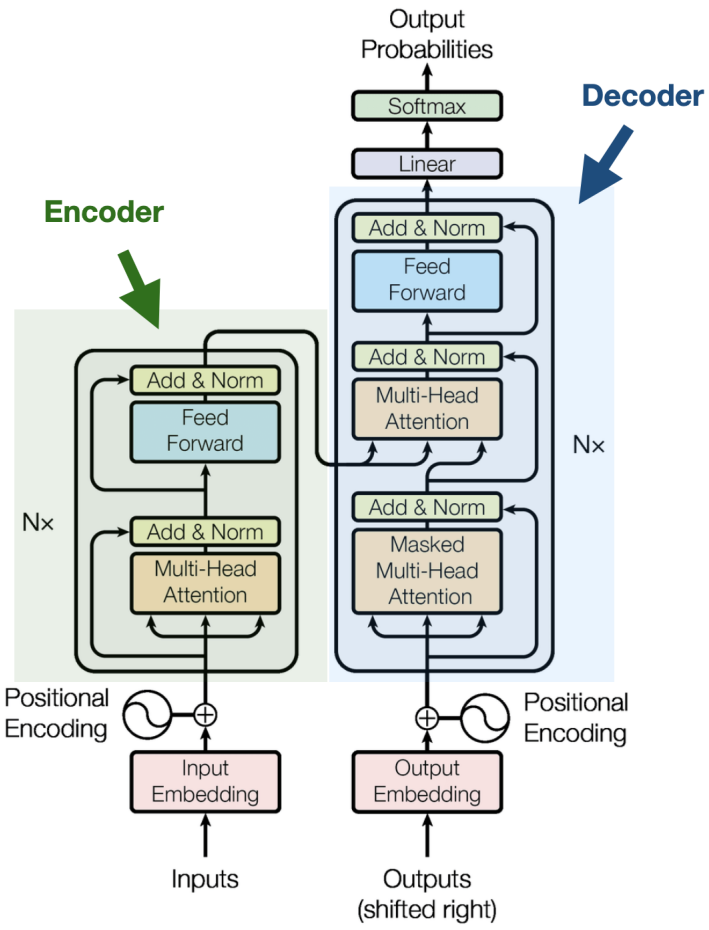


Figure 2.1: Transformer model architecture with highlighted encoder and decoder part, source:[23]

Encoder

The encoder (green part on Figure 2.1) consists of six identical layers stacked on top of each other, where each layer has two sub-layers. The first is a self-attention mechanism that relates different positions of a single sequence in order to compute representatio of the sequence [23]. The second is a simple, feed-forward fully-connected network consisting of two linear transformations. The attention layer helps the encoder look at other words in the input sentence to enhance some parts and diminish the others. These outputs are then fed to the neural network, which is independently applied to each position. This is what makes transformers parallelizable, in contrast to RNN.

Decoder

The decoder functions in a similar fashion to the encoder, except an additional attention layer is put, which instead captures relevant information from the generated encodings. It determines how input to the decoder and output from the encoder are related to each other. Next, the neural network makes content more digestible by the next decoder block or a final linear layer.

2.5.2 BERT

BERT [24] stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers and is a transformer-based language representation model introduced by Google in 2018. As explained in the original paper [24] "BERT's model architecture is a multi-layer bidirectional Transformer encoder based on original implementation described in [23]". This means that BERT, fundamentally is a stack of transformer encoders (see Section 2.5.1 and Figure 2.1). The original BERT-base model consists of 12 encoders with 12 self-attention heads, whereas its bigger version BERT-large uses 24 encoders with 16 self-attention heads.

Like the transformer model, BERT relies on an attention mechanism to generate high-quality, context-aware word embeddings. It differentiates itself from other language models like [25] or [26] by reading the text input sequentially and reading in both directions at once. The whole input to BERT needs to be given a single sequence. Special [CLS], [SEP] and [PAD] tokens are inserted by BERT to make sure the input is understood properly. As the original paper explains [24] "The first token of every sequence is always a special classification token [CLS]. The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks." This means that only this token is being looked at when using BERT for classification tasks. [SEP] token separates the end and the start of two sentences in the same sequence input. [PAD] enables BERT to receive a fixed length of sentence as input by padding shorter claims to the desired length. That's usually how data is prepared before the model is trained.

Because at a high-level BERT is a neural network architecture, additional neural layers can be inserted on top of it to train BERT for several natural language processing problems. This can be done through two phases:

Pre-Training Phase

The goal of pre-training is to make BERT learn what is language and context. It does so by training on two supervised tasks simultaneously: Masked Language Modelling (MLM) and Next

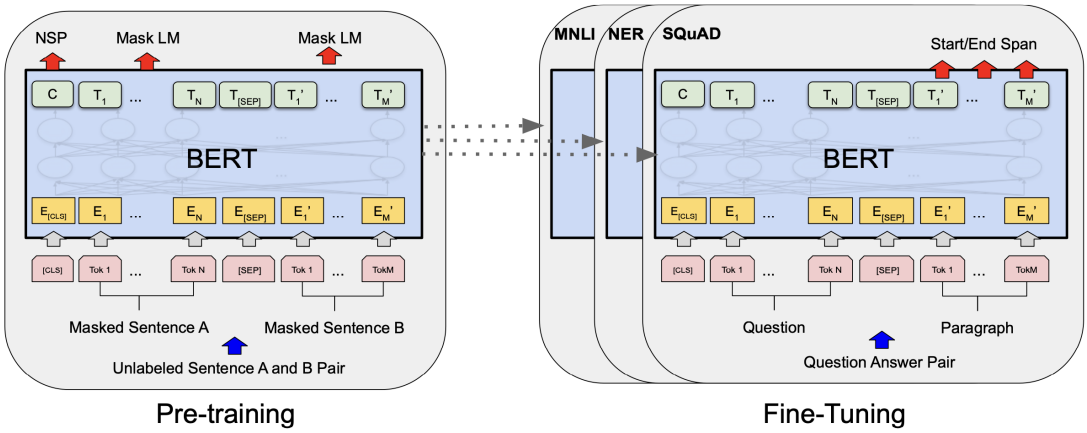


Figure 2.2: Pre-training and fine-tuning procedures of BERT. Source: [24]

Sentence Prediction (NSP). The input is a set of two sentences, where each token is a word that gets initially converted into already pre-trained embeddings. MLM hides 15% of input words and lets the model figure out what are the missing pieces by looking in both directions. In the context of NSP, BERT determines if the second sentence is the subsequent sentence in the original document. The original BERT model has been pre-trained on BooksCorpus (800M words) and English Wikipedia (2,500M words), therefore it is usually only the fine-tuning phase of the model that is required to train it for a particular task.

Fine-Tuning Phase

Fine-tuning phase allows us to further train BERT for specific NLP problems. It is straightforward since the self-attention mechanism in the transformer allows BERT to model downstream tasks. As described in the original paper[24] and [27] in fine-tuning, it is common to add one or more fully-connected layers on top of the final encoder layer. For classification problems supervised training can be then performed with a pre-trained model, additional classification layers and a labelled dataset. BERT fine-tuning is fast, as it is only the output parameters that are learned from scratch and the rest are slightly adjusted.

2.5.3 RoBERTa

RoBERTa [28] stands for **R**oboustly **o**ptimized **BERT a**pproach and is a model introduced by researchers from Facebook that suggests a series of improvements on top of the original BERT model. In particular the main differences are:

- **Dynamic masking** - BERT relies on static masking, meaning it performs it once during

data preprocessing. RoBERTa uses dynamic masking by generating the masking pattern every time a sequence is fed to the model.

- **Full sentences without NSP** - RoBERTa removes Next Sentence Prediction and instead "each input is packed with full sentences sampled contiguously from one or more documents, such that the total length is at most 512 tokens."[28]
- **Larger datasets** - RoBERTa is pre-trained on a larger dataset with varying domains like BookCorpus, CC-News, OpenWebText and STORIES.
- **Large vocabulary** - RoBERTa increased BERT English vocabulary from 30k to 50k words.

RoBERTa managed to outperform BERT on various natural language understanding benchmarks such as GLUE or SQuaD. Since the introduction of BERT, many RoBERTa-like models have been implemented by researchers to not only improve BERT's overall performance but also make them more task-specific.

2.5.4 Other BERT-based Models

BERT's architecture has proven to be successful in various NLP problems, hence many BERT-based models, sharing similar structures have been developed. Previously mentioned RoBERTa focuses on sheer improvement during training. ALBERT proposed in [29] aims to lower memory consumption and increase the training speed of BERT by reducing parameters, while achieving comparable results. Models like BERTweet [30] and exBAKE [31] modify the data used in the pre-training process, adding English tweets and news articles respectively. In this project, several already pre-trained BERT-based models are evaluated to compare their performance in classifying fake news.

2.5.5 Fake News Detection With BERT-based Models

Since its release, BERT has been used to attempt and solve several NLP classifications problems, including fake news detection. For example, researchers from [31] pre-trained their own BERT-based model with additional input data of 300k articles from CNN and Daily Mail to improve fake news articles detection capabilities. They use a simple linear layer and softmax for multi-class classification, with a weighted cross-entropy loss function during training to counter their imbalanced dataset. FakeBERT [32] is another misinformation classifier that as the authors say "combines different parallel blocks of the single-layer deep Convolutional Neural Network

(CNN) on top of BERT word embeddings". Surprisingly, given the nature of CNN and RNN, their model managed to outperform a recurrent neural network. Whereas previously mentioned approaches focus solely on word-embeddings, researchers from [33] managed to combine network and user-related features with the BERT encodings to classify fake or genuine COVID-19 related tweets.

2.6 Fake News Datasets

Regardless of whether the fake news classification problem is solved using traditional machine learning or deep learning methods, having a consistent, truthful and large-enough dataset is key to achieving accurate and unbiased results. This section shortly examines some of the publicly available datasets that can be used for misinformation classification. Their shortcomings in relation to this project are also stated.

The Table 2.1 summarises relevant information such as quantity, content type, annotator and labelling about popular and accessible fake/real news datasets. This representation is heavily inspired by work done in [34], where a similar analysis has been conducted. Because both [8] and [35] did not name their datasets, for purpose of this project these are titled TwitterFakeNews and CovidFakeNews respectively.

Initial manual inspection has also been conducted and summarized in Table 2.2, with the goal in mind to find an appropriate dataset for this project. It can be concluded that none of the datasets presents sufficient resources for training an unbiased, reliable ML model by itself. On one hand, big datasets like CREDBANK or TwitterFakeNews provide very noisy data with lots of false positives, due to the nature of automatic labelling and lack of verification. On the other hand datasets like PHEME, or BuzzFace might provide more accurate labelling, but their sizes make them unattractive. Kaggle Dataset or FakeNewsNet are mostly related to whole articles, making it hard to train for a short-text classification problem. Some datasets are also very specific: both CovidFakeNews and CoAID are concerned only with Covid-19, whereas FEVER entries are only formal, factual statements from Wikipedia. LIAR is one of the few datasets that shows promise due to its manual labelling process and good quality text data.

Dataset	Content	Size	Labeling	Annotator
CREDBANK [36]	Twitter posts related to news labelled as real or fake	60M	5 levels of truthfulness from 30 different annotators	Crowd-sourcing
BuzzFace [37]	Facebook posts and comments	2263	4 levels: agrees, disagrees, discusses and unrelated	Previously checked by Buzzfeed news agency
FakeNewsNet [38]	Whole articles and tweets sharing those articles	24.000	Binary: true or false	Previously checked by Politifact and GossipCop news agencies
LIAR [39]	Short, mostly political statements from various contexts	12.500	6 levels: true, mostly true, half-true, barely true, false , pants fire	Previously checked by Politifact news agency
FEVER [40]	Short factual statements from Wikipedia	180.000	3 levels: supported, disprovied, and not enough information	Trained human annotators
Kaggle Dataset [16]	Headlines and bodies of reliable/unreliable news articles	50.000	Binary: true or false	No information given
TwitterFakeNews [8]	Fake or true Tweets	200.000	Binary: true or false	Automatically labelled from previously picked trustworthy and non-trustworthy accounts
CovidFakeNews [35]	Social media posts and articles on COVID-19	10.000	Binary: true or false	Manual annotating done by a team
CoAID [41]	Diverse COVID-19 articles and social media posts	4250	Binary: true or false	No information given
PHEME [42]	Rumour and Non-rumour Tweets	330	Binary: true or false	Journalistic team and crowd-sourcing

Table 2.1: General information about publicly available fake news datasets.

Dataset	Initial Evaluation
CREDBANK	Low quality tweets (mostly spam) and a very small amount of fake entries. Collected tweets are just id's and would have to be scraped.
BuzzFace	Small dataset, where most posts were already deleted (cannot scrape).
FakeNewsNet	Articles themselves are too large for this project, however tweets sharing them might prove useful.
LIAR	Good quality manually labelled data of respectable size. No striking flaws at first sight.
FEVER	Factual-like statements that resemble Wikipedia facts. The data entries does not really relate to current style of news and headlines.
Kaggle Dataset	Headlines might prove to be useful, but lack of information about this dataset makes it less trustworthy
TwitterFakeNews	Has a good size and the claims are of desiarble length, however automatic labelling makes it hard to trust
CovidFakeNews	Although oriented only around COVID-19, the data seems quality.
CoAID	Simillar to CovidFakeNews.
PHEME	Very small dataset, oriented around rumours, which is a different thing than misinformation.

Table 2.2: Fake news public datasets with initial, manual data-quality evaluation.

2.7 AI Explainability

According to Wikipedia[1] "Explainable AI (XAI) is artificial intelligence, in which the results of the solution can be understood by humans. It contrasts with the concept of the 'black box' in machine learning where even its designers cannot explain why an AI arrived at a specific decision.". Effective AI models perform decisions that are usually very difficult to explain. However, in today's world proposing an accurate and efficient solution for detecting misinformation is no longer enough. As artificial intelligence becomes increasingly influential in our everyday lives, so does the responsibility of the developers to make trustworthy models. This section explains the most popular XAI approaches that can be applied to deep learning classifiers.

2.7.1 SHAP

SHAP [43] is a game-theory inspired method that attempts to enhance interpretability by computing the importance values for each feature for individual predictions. They are based

[1] `https://en.wikipedia.org/wiki/Explainable_artificial_intelligence`

on Shapley values [44], which decompose the final prediction into the average contribution of each attribute. This method can be used to give insights into the importance of features across both the whole model and a single prediction. However, due to their nature, it is usually used to explain models globally. As noted in experimentations conducted in [45], SHAP is a "slow and computationally expensive technique as it requires the Shapley values to be calculated for all features". This makes them undesirable for NLP tasks because each word is a feature in these models.

It does however provide meaningful insight when considering different feature types. Researchers from [9] focus their paper on analysing attribute contribution to machine learning models in fake tweet classification tasks, using SHAP explainability. They found most features to behave differently, based on the attribute group they were used in.

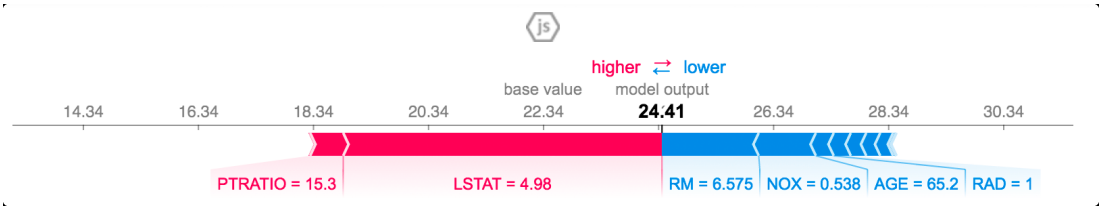


Figure 2.3: SHAP graphical explainability example

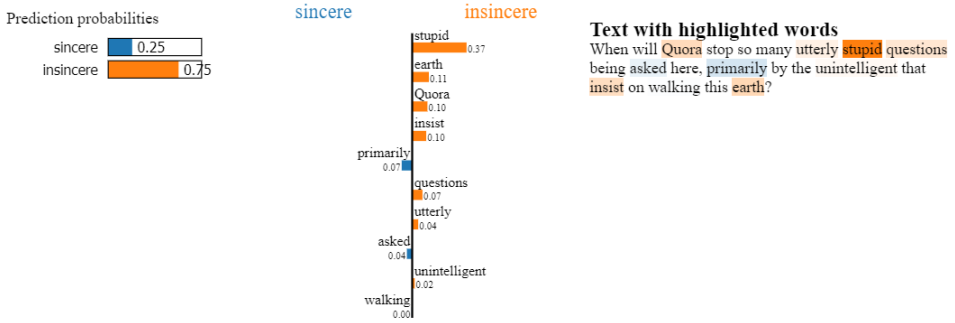


Figure 2.4: LIME graphical explainability example

2.7.2 LIME

LIME [46] is another popular XAI approach. The method is based on local surrogate models, which are trained to estimate predictions of black-box models. LIME trains local surrogate models, instead of the global one to explain individual output locally. It tests the results of the predictions when the black-box model is given variations of the input data. This dataset of variations and their output from the original model is then used to train an interpretable model (like a decision tree) to generate explanations. The explanations themselves reflect the

contribution of each feature to the prediction of a data sample. An example of such can be seen in Figure 2.4.

Researchers in [47] conduct a survey on human-interpretability of both LIME and SHAP Explainable AI techniques put on top of the BERT model. They found that LIME was ranked the highest overall and got significantly better scores than SHAP. The same can be said about findings in [48], where authors measure the stability of XAI methods and conclude that LIME is best suited for human interpretability.

Chapter 3

Requirements & Specification

This project's components can be categorised into 3 main sections: datasets, machine learning models and software. Upon successful completion, all three areas should be delivered, evaluated and combined in the context of fake news detection. Datasets are used to train, test and validate transformer-based classifiers. The models themselves serve as means for predicting claim veracity. The software allows for an intuitive inspection of the project's results. A very general description of this project's pipeline is shown in Figure 3.1. This chapter introduced a high-level overview of project components in the form of requirements, alongside their entailment to final deliverables. A more detailed description of how these components are achieved is laid down in later Chapters.

3.1 Dataset Requirements

As mentioned in Chapter 1, a big portion of achieving unbiased results is creating an appropriately rich dataset for training a machine learning classifier. The requirements below highlight the main deliverables of this project related to datasets.

D1 Analysis and formatting of existing datasets from Table 2.1, in order to inspect undesired characteristics and format them accordingly. This is done to create a new real/fake news dataset.

D2 Verification method to validate labels in automatically annotated or suspicious data entries.

D3 Main dataset for training machine learning misinformation classifiers, made using analysed datasets. This dataset should be split into training, validation and testing datasets.

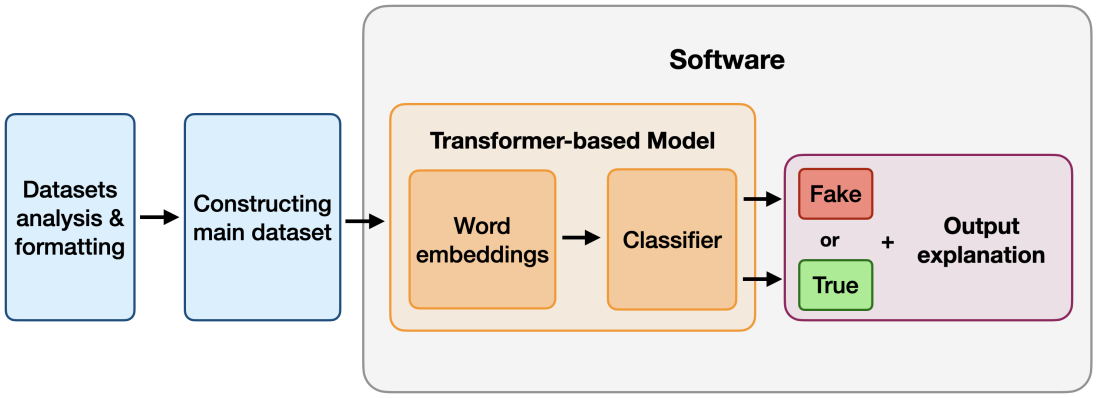


Figure 3.1: General pipeline of this project's deliverables.

3.2 Model Requirements

These requirements present main components related to transformer-based machine learning models that are implemented to classify claims based on their word embeddings.

M1 Universal transformer-based model architecture for fake news detection that creates appropriate word embeddings and feeds them into a neural network classifier.

M2 The trained model that reads a textual input claim and outputs a fake/true prediction with probability estimation.

M3 Experimentations with different neural network classifiers and transformer encoders for picking the best performing model.

M4 AI Explanation provided apart from the model's output for increasing trustworthiness and transparency of model's predictions.

3.3 Software Requirements

Software application serves as an easy-to-use tool for inspecting the main deliverable of the project: fake news classifier with AI Explanation put on top of it. To achieve this, the following requirements need to be delivered:

S1 The application should load an already trained model for misinformation classification. The model should not be reloaded every time its main page is visited.

S2 User should be able to type in a text and verify its truthfulness by using a loaded model.

S3 User should be prompt with according loading animations when the model is being loaded, or when the claim verifying process begins.

S4 The model's output and its AI Explanation should be displayed on the screen, once predicting process is completed.

S5 The application should provide an additional explanation, allowing a user to gain more insight into the inner working of the software/model.

Chapter 4

Design

This chapter builds on Requirements and Specification and describes the general approach for achieving the project's deliverables. It evaluates possible ways of producing them and explains the rationale behind chosen design decisions. This section starts with describing the methodology for picking and constructing datasets used in the implemented classifier - a topic of the next section. It finishes with the part about AI explainability and software.

4.1 Datasets

As mentioned in Section 2.6 most of the existing datasets for misinformation classification present themselves with issues such as small data size, inaccurate labelling, non-manual annotations, undesired claim length or topic-oriented data. However, many of these datasets also present promising features that can be used by extracting relevant data entries. Hence, this project aims to thoughtfully merge multiple existing datasets into one. This is done to increase the size, versatility and richness of the dataset, resulting in limiting its bias.

4.1.1 LIAR as Foundation

The LIAR dataset stands out from the rest, as it presents promising properties for ML modelling. Its statements are short, they come from multiple contexts (Figure 4.1) and are concerned with a variety of topics. Additionally, they were annotated by *Politifact*, which increases the trustworthiness of their labels. Because of that, **LIAR** is used as a foundation of a newly created dataset for this project. Moreover, datasets that were previously described as inadequate for training a classifier by themselves, are examined further to extract valid data entries.

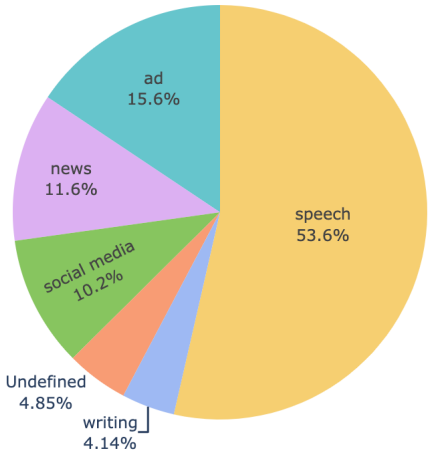


Figure 4.1: Distribution of claim contexts in LIAR dataset.

4.1.2 Other Datasets

From all datasets listed in Table 2.1, six of them were chosen for further analysis and formatting: FEVER, Kaggle Dataset, CovidFakeNews, CoAID, TwitterFakeNews and previously described LIAR. **FEVER** is a big dataset labelled with trained human annotators, hence it provides very reliable data. The nature of Wikipedia-like statements makes it less desirable for this problem, however, putting enough data entries should make the model more versatile. Because of their short length, claims from **Kaggle Dataset** fit the format of the project. Moreover, as described in Section 2.3.4, this data has already been used in many fake news detection approaches with great success. Both **CovidFakeNews** and **CoAID** are considered to help with classifying COVID-19 related news - a prominent subject in today's society. **TwitterFakeNews** dataset is a very big but also noisy dataset that was labelled automatically. Their data was chosen because of its size and promising results the authors managed to obtain with their fake news classifier. However, because of automatic annotations, only the best percentage of both true and fake data entries are considered. This is determined by using the Google Search Verification process 4.1.3.

4.1.3 Google Search Verifier

As stated in requirement D2, an additional process of verification is carried out to diminish the flaws of some datasets and is inspired by the Fact Verification Score from [33]. The authors use tweets as an input to a search engine and calculate the similarity score between the tweet and the results from reliable sources. This project uses an adaptation of their approach. From

initial testing and common sense, an assumption has been made that when the similarity is high, the claim is more likely to be true, and if it is low, it increases the chances of it being fake. If the sentence has a lot of comparable results it usually means it was discussed in the mainstream media. As an example, two alternately labelled claims got their similarity score calculated, with the true sentence ("Beyonce faces $20M copyright suit from YouTube stars estate") getting a much better score compared to fake news ("Site is getting blown up now thanks to Real Clear Politics"). Their google results can be seen in Figure 4.2. This tool can help deal with noisy and less reliable data entries.

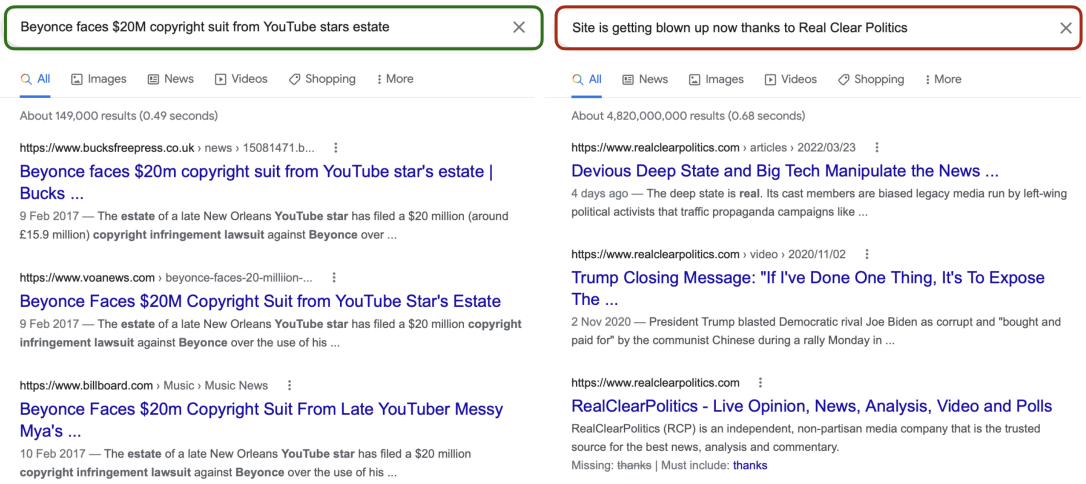


Figure 4.2: Google results of two example claims: one truthful (highlighted in green and one fake (highlighted in red).

4.1.4 Analysis & Formatting

Each dataset chosen in Section 4.1.2 is first analysed and formatted to increase their quality and perform necessary data cleansing. The analysis serves as a way to recognise the specific characteristics and flaws of each dataset. Its findings are used to implement appropriate formatting techniques for countering them. For instance, CovidFakeNews entries are often filled with hashtags or URLs. Hence, they need to be either removed or transformed in a way that does not change the meaning of the original claim. Additionally to dataset-dependent formatting, most of them go through common pre-processing techniques like unusual characters or duplicates removal. Furthermore, since around 2500 entries in the LIAR dataset are considered to be 'half-true', Google Search Verifier is used to label them as true or false.

4.1.5 Merging Datasets

The main dataset for training a machine learning classifier is constructed through merging data entries from analysed datasets. As stated earlier, all LIAR entries that were not deleted during data cleansing are used as a foundation of this new dataset. In contrast, only specific claims are merged from other datasets, where the data is first split based on claim length and its label. Next, a particular percentage of data entries is randomly chosen to make sure to not upset the balance of the dataset and to keep LIAR entries predominant. Furthermore, around 130,000 entries from TwitterFakeNews went through the Google Search Verification process, to find the claims that are most likely to be correctly annotated as true or fake.

4.2 Fake News Classifier

The fundamental part of this project is creating and training a fake news classifier. As discussed in Chapter 2 there exist many ways for implementing such a model. Traditional machine learning approaches presented in Section 2.2 have already been used extensively for this task, and leave little to no room for improvements and experimentations. Different types of neural networks (see 2.3.4) have also already been implemented for fake news detection. However, since transformer models (Section 2.5) solve the shortcomings of RNN and CNN in natural language processing tasks, a BERT-based model was chosen as a misinformation classifier. To transform BERT-like models into a classifier, an additional neural network is put on top of it. In short, the combined model takes as input predefined word embeddings, that get adjusted through the 'fine-tuning' phase, and are later categorised as true or false. Figure 3.1 serves as a visualisation of this process.

4.2.1 BERT-based Model

As already pointed out in Section 2.5.4, various BERT-like models have been developed since the release of the original BERT. Most of them try to improve their overall pre-training and fine-tuning procedure or were created with a specific task in mind. Because many of these models are already pre-trained and are easily accessible, several of them are used for experimentation and evaluation purposes. The end goal is to find the best performing classifier. The list of considered pre-trained BERT-based models can be seen in Table 4.1

In total, eight different BERT-like models are used in the project. The base and large variations of the model indicate the number of the model's parameters and its size. For example,

Pre-trained model	Letter Casing	Pre-training data used
BERT-Base	Uncased	BookCorpus, Wikipedia
BERT-Base	Cased	BookCorpus, Wikipedia
BERT-Large	Uncased	BookCorpus, Wikipedia
BERT-Large	Cased	BookCorpus, Wikipedia
RoBERTa-Base	Cased	BookCorpus, Wikipedia, CC-News, OpenWebText, STORIES
RoBERTa-Large	Cased	BookCorpus, Wikipedia, CC-News, OpenWebText, STORIES
ALBERT Base v2	Uncased	BookCorpus, Wikipedia
BERTweet	Cased	English Tweets (Including COVID-19 related)

Table 4.1: Pre-trained BERT-based models evaluated in this project

the BERT-Base model consists of 12 encoders with 12 self-attention layers, whereas BERT-Large has 24 encoders with 16 self-attention heads. Moreover, 'uncased' and 'cased' labels imply the ability of the model to recognize cased characters. BERT and RoBERTa models were chosen, as they are the most popular and widely considered the best among transformer encoders. ALBERT could be the correct model for the software due to its lightweight implementation, whereas BERTweet was chosen as most of the datasets consists of social media posts. A general architecture with additional neural network layers is created for all of the models and can be seen in Figure 4.3.

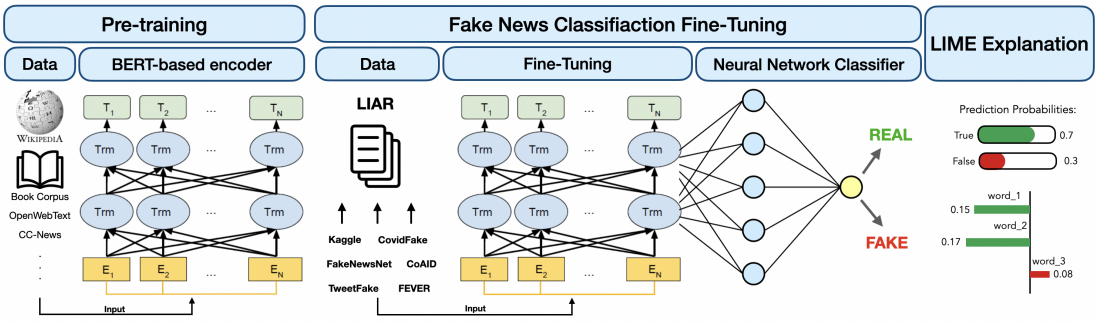


Figure 4.3: BERT-based model architecture with neural network classifier and LIME explanation.

4.2.2 Neural Network Classifier

It is very common to use additional artificial neural network (ANN) layers for text classification on top of BERT-based models. This is because transformer-based models follow the same architecture of a neural network with an addition of attention heads. As described in Section

2.5.5, it is also possible to apply more complex deep learning nets like CNN or RNN. However, the authors of BERT, RoBERTa have used a simple ANN on top of their encoders to set benchmark accuracy scores in NLP tasks. In fact, researchers in [49] conducted experiments on additions of RNN on top BERT models for SQuAD benchmark tasks. They saw a worse performance when applying deep learning methods instead of ANN. These findings, and the fact that BERT-based models are already more complex and effective architectures for NLP tasks (they should be able to learn anything that CNN and RNN do), is why a simple ANN with 1 hidden layer is used as a classifier in this project. Having said that, many different neural networks are used, in order to find the best-performing one. Different activation functions, the number of nodes, as well as hidden layers are evaluated.

4.2.3 Model Training

Once the model architecture is established, the training loop is used to fine-tune the models. Necessary components for such training consist of a tokenizer, an optimizer, learning rate, batch size and number of epochs. The authors of BERT recommend to chosee from the following values (from Appendix A.3 of BERT paper [24]):

- Optimizer: Adam
- Learning rate : 5e-5, 3e-5, 2e-5
- Batch size: 16, 32
- Number of epochs: 2, 3, 4

A tokenizer handles most of the parsing and data preparation needed for feeding data entries into the model. Every pre-trained model has its own implementation of the tokenizer that is used during fine-tuning.

4.2.4 Experimentation

To implement requirement M3 and pick the best performing classifier for software implementation, a necessary experimentation phase needs to be conducted. To do that, this project evaluates multiple neural networks and discussed BERT-based models, with popular machine learning metrics. These evaluation techniques include:

- Accuracy: A ratio of correctly predicted observations to the total of observations.

- Recall: Ratio of correctly predicted positive observations to all observations. This metric performs well when the costs of false negatives are high.
- Precision: Talks about how precise the model is out of positive predictions. This metric performs well when the costs of false positives are high.
- F1 Score: A weighted average of precision and recall. Takes both false positives and false negatives into account.
- ROC Curve: A graph that shows how much is the model capable of differentiating between labels in a binary classification problem. It measures the degree of separability.
- Confusion Matrix: It is a technique for summarizing the performance of a classification algorithm and can help decide on what types of errors the classifier is making.

At first, several different BERT-like model architectures with a simple ANN on top are implemented and evaluated, to find the best performing transformer-model. Next, to improve the final accuracy, distinct neural networks are tested on top of a BERT model with the highest score. Such an approach was chosen to reduce the number of possible combinations between neural networks and transformer models. The project does not possess enough GPU, time and memory power to evaluate all feasible pairings.

4.2.5 LIME AI Explainability

To make the model more trustworthy for potential users, additional explanations and insight are provided alongside the prediction itself. Hence, the model is required to output an explanation based on the individual input. Chapter 2.7 describes two of the more popular XAI approaches for deep learning models. Both LIME and SHAP can provide local explanations. However, as stated in [2.7.2], LIME is considered to be best suited for human interpretability therefore, it was chosen for this project. It outputs the probabilities of true/false prediction, as well as, the weight of each word on classification and displays it in an understandable graphic.

4.3 Web Application

An intuitive, easy-to-use web application is one of the goals of this project and the development process concentrated on providing an intuitive user experience. Its main goal is to fulfil Software Requirements (S1-S5 in Section 3.3), wrap around the system architecture shown in Figure 3.1, serving as a tool for fake news classification. There are several convenient frameworks for

deploying machine learning models online using Python, hence, a web application was chosen instead of a desktop application.

Chapter 5

Implementation

The implementation chapter follows the chronological order of the development process for this project. It starts with explaining the methodologies concerned with datasets - from initial analysis and formatting until the final construction technique. It then follows with an in-depth explanation of BERT-like models implementation, alongside their evaluation processes. It finishes with engineering both LIME explainability and the final web application. All of the implementation described in this Chapter can be seen in Appendix C.

5.1 Google Search Verifier

In an effort to verify annotations of particular claims, this project implements a Google Search Verifier described briefly in Section 4.1.3. The algorithm inputs the claim into a google search engine and scrapes the titles from the front page results. Next, it measures the similarity between the claim and the results headlines using *Levenshtein distance* - a metric that represents a minimum number of single-character edits required to change one string into the other. The script follows the pseudo-code in Algorithm 1 and its implementation is shown in Appendix C.1.1. Google uses a range of defensive methods that makes scraping their results challenging. To counter that a simple 5 second wait time in between queries is added and solved the issue in most cases. Additionally, the code uses random user agents each time to avoid getting blocked by web servers. The script was run in the background on Google Cloud Platform, whilst other datasets have been examined and formatted. It is worth noting that the algorithm calculates just the score, and the search results themselves are not saved anywhere.

Algorithm 1 Google Search Verifier Script

Require: $UG : useragentlist, DF : dataset$
  for `each` $title$ `in` DF **do**
    `Create google query with` $title$
    `Wait 5 seconds`
    `Send request with random agent` UG
    `Parse the titles to titles`
    $levenScore \leftarrow 0$
    $results \leftarrow 0$
    for `each titles` **do**
      $levenScore \mathrel{+}= Levenshtein(query, title)$
      $results \mathrel{+}= 1$
    end for
    `Save` $levenScore \div results$
  end for

5.2 Datasets Formatting

Insightful dataset formatting for eliminating bias and offsetting suspicious characteristics of each dataset is the central part of the project; hence, it started when development began. With the goal in mind to gather data of the highest quality, each dataset has been individually analysed in separate Jupyter Notebooks[1] (Appendix C.2). Common data science libraries were used such as Pandas[2], NumPy[3] for data manipulation and Matplotlib[4] and Plotly[5] for data visualisation. For finding particular patterns within the claims, the Python RegEx technique was applied.

General Techniques

Whereas this project conducts an individual analysis of each dataset, it also applies common pre-processing techniques to all datasets. The main methods used are:

- **Duplicates removal** - Surprisingly, many datasets include duplicate claims; thus, double entries were deleted.
- **Removing/Formatting claims with special characters** - Every dataset includes entries with unwanted characters such as '@', '#', '&amp', non-ASCII characters and emojis. Depending on the size of the dataset and its quality, these claims were either deleted or formatted.

[1] `https://jupyter.org`
[2] `https://pandas.pydata.org`
[3] `https://numpy.org`
[4] `https://matplotlib.org`
[5] `https://plotly.com`

- **Removing/Formatting claims with links** - Depending on the position of the URL within the claim, the links were also either deleted, or the whole claim was discarded.

5.2.1 LIAR

Considering that LIAR was chosen as the foundation of a newly created dataset, the goal was to inspect its entries thoroughly and avoid unnecessary data pruning. During analysis, LIAR was identified with two main issues:

1. Because the dataset had a 'speaker' column, some of the claims described what was said by the speaker, without specifically including him in the statement. An example of such a claim would be:

   "Says John McCain has done nothing to help the vets."

   The correct way to rephrase it would be to either include the speaker (a) or focus on the claim and delete the word 'Says' (b):

   (a) "Donald Trump says John McCain has done nothing to help the vets."

   (b) "John McCain has done nothing to help the vets."

   Because the project aims to fact-check the claims, instead of focusing on who was the speaker, wherever possible such sentences were transformed into the impersonal form shown in (b). Therefore, the claims starting with 'Says that', 'Says a', or 'Says the' were appropriately edited to reflect the transformation in point (b). However, many claims could not be altered in such a way, due to their personal essence. For example, removing 'Says his' from:

   "Says his political opponents brought 100,000 protesters into our state."

   would result in a claim that does not make much sense. Hence, claims starting with 'Says' and any of 'his', 'him', 'he', 'hes', 'she', 'shes', 'her', 'her' followed a transformation described in point (a). The examples of these transformations can be seen in Table 5.1

2. LIAR dataset collected news and labels from *Politifact* website, which considers six annotations for truthfulness ratings (levels from false to true): *pants-fire, false, barely-true, half-true, mostly-true, true*. Because this project implements a binary classification, the

Initial Claim	Formatted Claim
Says her accomplishments include a fiscally responsible budget agreement that controls state spending.	Chris Telfer says her accomplishments include a fiscally responsible budget agreement that controls state spending.
Says a highway was closed in El Paso because of bullets flying across the border	Highway was closed in El Paso because of bullets flying across the border.
Says he couldnt keep a 42-inch redfish he caught because of fishing rules.	Adam Putnam says he couldnt keep a 42-inch redfish he caught because of fishing rules.

Table 5.1: Example claim formatting performed in LIAR datasset.

labels *pants-fire, false, barely-true* are changed to *false*, whereas *mostly-true, true* to *true*. ***Half-true*** entries present themselves with an issue as they are hard to classify. Because they take up around 20% of the dataset, discarding them was very unattractive. Instead, an already described Google Search Verifier was used to determine their claim veracity. After their average Levenshtein score was calculated and adjusted for the claim character count, the entries were split into 4 different groups based on their length. Next, half of the best scores from each set were classified as *true*, while the other half as *false*. Albeit a simple approach, it provides an additional level of verification.

5.2.2 CoAID

The CoAID dataset has a peculiar issue in its dataset. Over 5000 of its entries are labelled as true, but the claims themselves are questions, and the majority of them cannot be answered with a simple 'true' or 'false', for example:

"What is the best household disinfectant for surfaces?"

Hence from 6000 entries, only around 1000 are taken, with most being false entries.

5.2.3 CovidFakeNews

Because the CovidFakeNews dataset consists solely of social media posts and articles on Covid-19, the project investigates the placement of 'hashtags' (#) and 'mentions' (@) - a common ingredient of tweets. The main finding was that many entries end with links to some external sources, mentions, hashtags or unusual characters. An appropriate function was implemented to delete these unnecessary characters from the back of a claim. Example transformation are presented in Table 5.2 under 'Rule' number 1. Furthermore, additional formatting was done,

to ensure that hashtags were transformed into normal words or deleted ('Rule' number 2). In the end, the data proved noisy, and around 3000 entries were permanently dropped.

Rule	Original Claim	Formatted Claim
1	We just announced that the first participants in each age cohort have been dosed in the Phase 2 study of our mRNA vaccine (mRNA-1273) against novel coronavirus. Read more: https://t.co/woPlKz1bZC #mRNA https://t.co/9VGUoJu5cS	We just announced that the first participants in each age cohort have been dosed in the Phase 2 study of our mRNA vaccine (mRNA-1273) against novel coronavirus.
1 & 2	WHO clarifies remarks on asymptomatic #coronavirus as rare. https://t.co/v6HSi92aGo	WHO clarifies remarks on asymptomatic coronavirus as rare.

Table 5.2: Example claim formatting performed in CovidFakeNews dataset.

5.2.4 FEVER

Analysis of FEVER was straightforward, as the claims are formed directly based on Wikipedia articles and do not have any strange characters a social media post would contain. Thus, it required only common formatting techniques. It is worth noting that around 40000 rows were discarded at the start because they were classified as 'not enough info'.

5.2.5 Kaggle Dataset

Numerous Kaggle dataset claims consist of unusual additional information that was put in the squared brackets at the end of the claim (keywords like Video, Tweet, Photo etc.). Moreover, some of the entries contain meaningless words ('factbox', 'wow!', 'exclusive!', 'breaking' etc.), which do not add anything valuable to the text. These characteristics were primarily found in fake tweets; hence, not deleting them would create a bias in the dataset. In order to offset the negative effects of such statements, formatting with RegEx was done to find appropriate patterns and substitute them with suitable replacements. Examples of modified claims are shown in Table 5.3

Original Claim	Formatted Claim
FACTBOX: Oil companies in Caribbean, southeast U.S. continue restart after Irma	Oil companies in Caribbean, southeast U.S. continue restart after Irma
WOW! DONNA BRAZILE Tells Critics Of Her Book On DNC To 'Go To Hell' In Exclusive Interview [Video]	Donna Brazile Tells Critics Of Her Book On DNC To 'Go To Hell' In Exclusive Interview.

Table 5.3: Example claim formatting performed in Kaggle datasset.

5.2.6 TwitterFakeNews

To recall, the TwitterFakeNews dataset was scraped and labelled automatically from various trustworthy or untrustworthy social media accounts. This makes the collected data noisy and unreliable. Because the dataset is very big and only a small percentage of its entries are used during dataset construction, any suspicious characteristics led to the complete removal of the claim. This allowed to discard over 70k entries with unwanted keywords, links, etc. Following that, the remaining 110k claims went through a verification process using the Google Search Verifier script. After the results were calculated, they were later adjusted for their claim length to diminish the correlation between high similarity score and short sentence length. This resulted in a clear indication of data reliability and was used to pick claims with the best score.

5.3 Dataset Construction

The dataset construction process merges entries from previously formatted datasets. LIAR was not the only dataset used in its entirety, as both CovidFakeNews and CoAID presented good quality data after deleting and formatting flawed examples. Claims used from other datasets had to be limited, to keep LIAR as a major part of the final data. In order to pick these entries, the particular datasets were split into groups based on the character length. Afterwards, a percentage of either random (in the case of FEVER and Kaggle), lowest Levenshtein score (True entries of TwitterFakeNews) or worst Levenshtein score (False entries in TwitterFakeNews) were picked to be included in the final dataset. The ultimate dataset consists of 38,659 labelled claims with a moderately even distribution between true and false entries. Table 5.4 summarizes the sizes of particular datasets throughout the process of analysing and picking adequate claims. Figures 5.1 and 5.2 serve as a visualisation of the distribution between different data origins and the labels in the final dataset.

Dataset	Original size	Size after processing	Entries used (true, fake)
LAIR	12791	12506	12506 (5729, 6777)
CovidFakeNews	10700	7512	7512 (3167, 4345)
CoAID	5975	998	998 (46, 952)
FEVER	145449	102292	4000 (2000, 2000)
Kaggle Dataset	44898	37724	6400 (3200, 3200)
TwitterFakeNews	188773	111234	8247 (5000, 3247)

Table 5.4: Summary of datasets original size, size after processing and entries used in the final dataset.

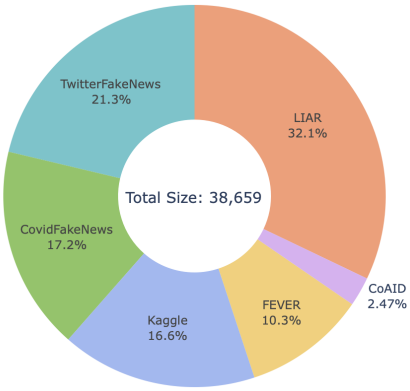


Figure 5.1: Distribution of data entries and their origins in the final dataset.

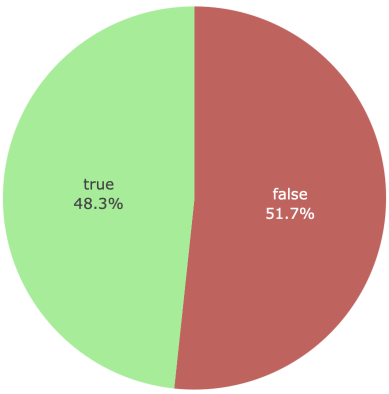


Figure 5.2: Distribution of true/false data entries in the final dataset

5.4 BERT-based Model Implementation

Implementing and evaluating BERT-based models was a focus of the second part of the project. There exist 2 popular libraries for implementing deep learning components - TensorFlow[6] and Pytorch[7]. This project uses Pytorch because it provides a more friendly user and debugging experience for Python developers. Moreover, many already pre-trained Transformer models components were imported from the powerful, open-source HuggingFace [8] library using their *from_pretrained()* function. This allowed for faster training times, as the pre-training phase of BERT-like models and their tokenizers has already been done by their community. Own

[6] `https://www.tensorflow.org`
[7] `https://pytorch.org`
[8] `https://huggingface.co`

implementation, training and evaluation were programmed in Google Colab[9] because they provide free access to GPUs, which dramatically speeds up the training time of Deep Learning models.

This section explains in detail how data preparation techniques, choosing hyperparameters, model architecture, training and testing were all implemented, using the tools listed above.

5.4.1 Code Structure

All the main code related to creating, fine-tuning, evaluating and experimenting with different BERT-based models is implemented in one Jupyter Notebook called 'Models_Implementation.ipynb' (See Appendix C.3.1). The whole file is very organised, with sections from 1 to 6 having all the necessary functions and model architectures for conducting later experiments. Sections 7 to 10 call these functions to define best-suited hyperparameters, evaluate different BERT-based models and neural networks. Section 11 serves as early experimentation with the LIME explanation that was later implemented in the software. The code is structured in such a way to support easier reusability, limit code repetition and allow for straightforward usage of these functions in the software implementation. All the code implementing Sections from 5.4 to 5.6 can be inspected in this file. The cell outputs were cleared to ensure the code is more readable. Section '7. Defining Hyperparameters' is the exception, because its output is needed to understand the hyperparameters chosen for training the models.

5.4.2 Data Preprocessing

The required data processing for inputting claims into BERT-like models starts with the Tokenizer. Each BERT model has its implementation of the Tokenizer, which satisfies all the following formatting requirements of BERT:

- Add special [CLS] (classification) and [SEP] (separation) tokens to the start and end of each sentence.
- Pad and truncate all sentences to a single fixed length.
- Explicitly differentiates real tokens from padding tokens (PAD) with the attention mask.

This process in the done by the *data_tokenization()* function. The tokens are pre-trained BERT word embeddings that are additionally fine-tuned during training. An example of a tokenized

[9] `https://colab.research.google.com`

sentence can be seen below. Unrecognised words are split into chunks that the tokenizer's vocabulary is able to identify. In this case, the name of the stadium 'Anfield' is split into 'An' and 'field'.

Original	Liverpool supporters gather at Anfield to celebrate
Tokenized	['Liverpool', 'supporters', 'gather', 'at', 'An', '##field', 'to', 'celebrate']
Token IDs	[5555, 6638, 8422, 1120, 1760, 2427, 1106, 8294]

Table 5.5: An example of a tokenized sentence and BERT's methodology to tokenize unrecognised words

Next data loaders are created and loaded with batches of data samples. Data loaders provide an iterable over a given dataset, making them more memory-efficient. They also include a sampler that makes sure the data is either accessed randomly or sequentially - an important part for comparing results over multiple models

5.4.3 Model Architecture

BERT, RoBERTa, ALBERT and BERTweet main components are all parts of the same Transformers class, implemented by the HuggingFace library that also stores already pre-trained models listed in Table 4.1. This allowed to construct the same structure of model architecture for all of them and focus on fine-tuning the model. The main difference in each model is the use of a specific Tokenizer and a different name. For clarity, this section explains a BERT architecture, however same can be applied to other models.

The classification model can be described through 4 layers: BERT-embedding layer, dropout layer, hidden layer and output layer. The embedding layer is the actual transformer model, whereas dense layers are hidden and the output layers of the ANN. Dropout randomly deletes certain nodes to prevent overfitting. In the code itself (see Section 5.1), all 3 layers after the embedding layer are combined as a 'classifier' as this is the actual component that classifies sentences. Because it is a feed-forward neural network, the 'Sequential' module from the Pytorch library has been used (the flow goes in one direction).

BertForFakeNewsDetection class also contains a *forward()* function that computes logits - an output of a function that maps values from -infinity to +infinity to the values from 0 to 1. More precisely the function feeds BERT the input to and retrieves the CLS token. This CLS token is then input to the classifier itself, which outputs the probabilities of classification.

Listing 5.1: BERT model architecture class. The version with more in-depth comments can is shown in Appendix C.3.1 Section 4.

```
class BertForFakeNewsDetection(nn.Module):
  def __init__(self, freeze_bert=False, bert_name='bert-base-cased'):
    super(BertForFakeNewsDetection, self).__init__()
    self.bert = BertModel.from_pretrained(bert_name) # BERT Embedding layer
    input_size = self.bert.config.to_dict()['hidden_size']
    self.classifier = nn.Sequential(
        nn.Dropout(0.2),           # Dropout Layer
        nn.Linear(input_size, 50), # Hidden Layer
        nn.ReLU(),                 # Activation Function
        nn.Linear(50, 2),          # Output Layer
        nn.LogSoftmax(dim=1)       # Activation Function in the output
    )
    if freeze_bert: # Freeze BERT if you do not want to fine tune it
        for param in self.bert.parameters():
            param.requires_grad = False
  def forward(self, input_ids, attention_mask):
    _, last_cls_token = self.bert(input_ids = input_ids, attention_mask =
        attention_mask, return_dict = False)
    logits = self.classifier(last_cls_token)
    return logits
```

5.4.4 Model Hyperparameters

The main optimization of the model's training process comes from defining its hyperparameters. Although there exist many different arguments that can be adjusted this project focuses on six:

1. **Loss function:** The loss function that was mainly used is the Binary Cross Entropy function, as it is widely implemented for binary classification problems.

2. **Optimizer:** To recall, the optimizers are algorithms used to minimize the loss function during training. The optimizer recommended by the authors of BERT is Adam Optimizer. Since the release of the paper, a new version called AdamW came out. AdamW solves issues related to calculating the weight decay and the authors themselves showed experimentally that AdamW yields better training loss and overall results than Adam.

Hence AdamW optimizer has been used across experiments.

3. **Learning Rate Scheduler:** As said in (design hyperparameters) one of the proposed learning rates was used in this project. Specifically 2e-5 rate since it was yielding better results than 3e-5 or 4e-5 in the initial training.

4. **Max input length:** Because each sentence in BERT needs to be either pad or truncated to the same input length this has to be specified as one of the hyperparameters. Figure 5.3 shows the dataset's claims length distribution after tokenization. Although the maximum tokenized input is 105, it's a clear outlier in the dataset. Therefore, the maximum length was set to 80, as the project focuses on shorter claims. This also leads to faster training times.

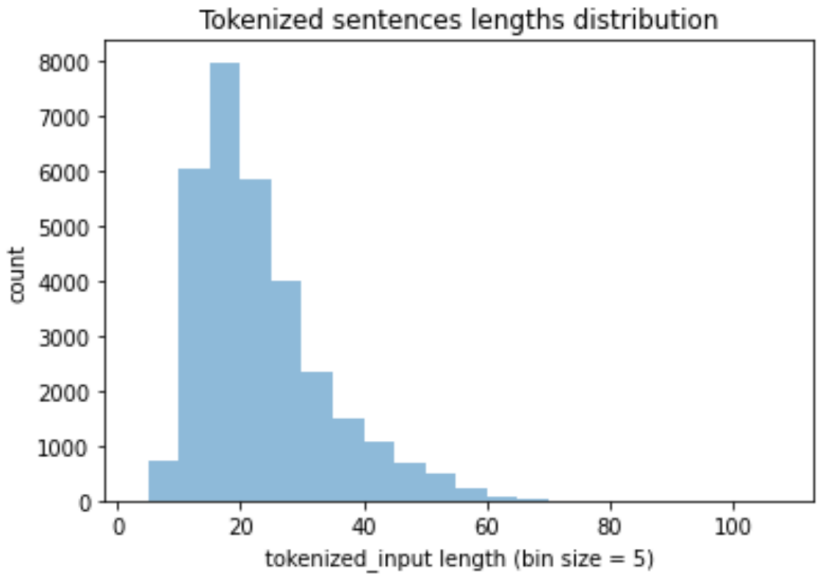


Figure 5.3: Distribution of tokenized claims in the final dataset.

5. **Batch size:** Batch size specifies the number of samples that will be propagated through the network before updating the internal model parameters. Because usually a larger batch size is recommended for Transformer models [50], and the BERT paper recommends 16 or 32, the latter one was chosen for this project.

6. **Epochs:** The number of epochs is the number of complete passes through the training dataset. To determine the optimal number o epochs for this project, initial testing has been done to track both training loss and validation loss (using a separate validation dataset). The results are presented in Table 5.6 and the full procedure can also be seen in Appendix C.3.1 section inside a '7. Defining Hyperparameters' chapter. The training

loss goes down with each epoch, suggesting that the network keeps on learning. However, after the second epoch, the validation loss goes up from 0.357 to 0.409, which means that the model is not really learning, but instead overfitting on the dataset. Hence for the rest of the experiments, the epoch number o 2 was chosen, as the optimal number of iterations before the model overfits on the given data.

Epoch	Train loss	Val loss	Val Accuracy	Val F1	Val Recall
1	0.436	0.366	0.82	0.80	0.78
2	0.309	0.357	0.82	0.81	0.79
3	0.222	0.409	0.82	0.81	0.84
4	0.154	0.487	0.82	0.81	0.83

Table 5.6: Results from initial training and evaluation for determining number of epochs. 'Val' rows represent metrics related to the validation dataset

5.4.5 Training Loop

The main component of creating and fine-tuning the model is the training loop itself. The function requires the model based on its class architecture shown in 5.1, data loaders and other hyperparameters to be defined beforehand. The full code for it can be seen in Appendix C.3.1 in '5. Training-Loop' section. Its pseudo-code with explanations describing the steps is represented in Algorithm 2. For each epoch, the model is trained and its performance can be evaluated on the validation dataset (if do_evaluation is True). The code follows the general technique in training neural networks described in Section 2.3.1.

5.4.6 Testing

The testing loop follows a very similar algorithm to the evaluation part of the pseudo-code in Algorithm 2. Testing is done on an independent dataset to ensure that model has not seen the claims beforehand. The testing loop does not require epochs, an optimizer or a learning rate scheduler because it should not involve the backpropagation part of the initial training. The gradients are static and the model is put in evaluation mode. The results are stored in arrays and are used to evaluate the model's performance.

5.4.7 Evaluation

To evaluate the performance of implemented models, common machine learning metrics described in Section 4.2.4 were used. All accuracy, recall, precision, f1 score, confusion matrix

Algorithm 2 BERT training loop

Require: $Model$, Tr_DL : *train data loader*, Val_DL : *validation data loader*, Df : *dataset*, Opt : *optimizer*, Sch : *learning rate scheduler*, *do_evaluation*

```
for each epoch do
    Put the Model in training mode
    for each batch in Tr_DL do
        Unpack the batch from Tr_DL and load its data into GPU
        Clear any previously calculated gradients
        Perform a forward pass to compute logits and loss
        Perform a backward pass to calculate gradients
        Update parameters of the Model by using the Opt
        Update the learning rate of the Sch
        if do_evaluation is True then
            Put the Model in evaluation mode
            for each batch in Val_DL do
                Unpack the batch from Val_DL and load its data into GPU
                Perform a forward pass to compute logits and loss
                Get Model predictions and compare with actual labels
            end for
        end if
    end for
end for
```

and roc curve were calculated using *sklearn* Python library that provides a *metrics* module for this specific purpose. Additionally, the training loop tracks training loss and time elapsed since the start.

5.5 Models Experimentation

As discussed in Section 4.2.4 BERT-based models from Table 4.1 are evaluated on the training and testing datasets, to pick the best-suited model for the web application. The experimentation is conducted in two phases: BERT-based model evaluation with default neural network and neural networks evaluation with the best BERT-based model. In both cases, each model is trained from scratch using the training loop (5.4.5), tested with the testing loop (5.4.6) and evaluated with the implemented metrics (5.4.7). Before each experiment a *set_random_seed()* function is called that sets a random seed in each of the libraries that are used in this process. This is made to ensure the results are reproducible and comparable between models.

BERT model	Accuracy	F1 Score	Recall	Precision	Training Time	Model Size
bert-base-uncased	0.796	0.793	0.805	0.781	27 min	438 mb
bert-base-cased	0.824	0.817	0.813	**0.821**	28 min	433 mb
bert-large-uncased	0.803	0.804	0.833	0.777	89 min	1.34 gb
bert-large-cased	0.825	0.821	0.828	0.814	87 min	1.34 gb
roberta-base	0.831	0.832	**0.864**	0.802	30 min	500 mb
roberta-large	**0.840**	**0.839**	0.**864**	0.816	46 min	1.42 gb
bertweet-base	0.834	0.832	0.853	0.813	**15 min**	540 mb
albert-base-v2	0.792	0.782	0.767	0.797	27 min	**47 mb**

Table 5.7: BERT-based models testing results.

5.5.1 BERT-based Models Results

Because the code for creating, training and evaluating BERT-based Models and their components is constructed from functions, all of the models can be trained within a single for-loop. Hence the '10. BERT Experiments' section in the notebook (see Appendix C.3.1) utilises one loop to do all previously-discussed necessary steps. These include data tokenization, initialising training components, training the model, saving it, testing, evaluating and saving the results. The end results of these models are presented in Table 5.7.

The results show that RoBERTa models outperform any other BERT-based models in every metric, proving their high capabilities for NLP tasks. The large version, while yielding superior results in most categories, carries the largest model size of 1.42 gigabytes. This makes it less desirable for software implementation. Bertweet-base, which is also based on the training procedure of RoBERTa showed promising results that match the roberta-base model. This suggests that constructed dataset has a lot of resemblance to tweet-like claims. Roberta-base, being a smaller version of the best-performing roberta-large, also achieved high scores and even matched its bigger version in recall. It is also both considerably smaller in size and faster. Because of that, it was chosen to be included in further experiments and the web application. BERTweet model could have also been chosen; however, the final software is not intended only for verifying tweets. The more in-depth analysis of these results is conducted in Chapter 6 and the full testing results, including ROC curves and confusion matrices, are shown in Appendix A.1.

5.5.2 Neural Networks Evaluation

In order to achieve higher results, an additional evaluation of different neural network classifiers on top of the picked roberta-base model has been done. Because a simple ANN architecture was chosen, the only real components that could have been adjusted are the number of hidden layers, number of nodes and activation functions. Hence, the 'classifier' component of the BertForFakeNewsDetection class (Listing 5.1) was changed to the following feed-forward neural networks:

Layer	**NN-1**	**NN-2**	**NN-3**	**NN-4**	**NN-5**
Dropout	Dropout(0.2)	Dropout(0.2)	Dropout(0.5)	Dropout(0.2)	Dropout(0.2)
Hidden	Linear (768,50)	Linear (768,50)	Linear (768,256)	Linear (768,50)	Linear (768,50)
Activation Function 1	ReLU()	LeakyReLU()	ReLU()	GELU()	SELU()
Output	Linear(50,2)	Linear(50,2)	Linear(256,2)	Linear(50,2)	Linear(50,2)
Activation Function 2	LogSoftmax	LogSoftmax	Softmax	LogSoftmax	LogSoftmax

Table 5.8: Representation of the architecture of different neural network classifiers that were evaluated on top of roberta-base model. Each column represents one neural network.

Most of the changes to the neural network architectures come from applying different activation functions. The ones mentioned in 2.3.1 were used. 'NN-3' have a different number of nodes in the hidden layer and a different dropout ratio compared to other. The evaluation results of these nets with the roberta-base model are represented in Table 5.9. The differences in performance between these models are very slight, and they all perform similarly across all categories. The experiments suggest that the classifier itself has little power over a much more complicated BERT transformer model architecture, and is not worth tuning much further. In the end, neural network number 1 ('NN-1') was chosen since it achieved the best F1 score, and because ReLU is widely used across the field. All neural network architectures and testing results are present in Appendix A.2.

Layer	Accuracy	F1 Score	Recall	Precision
NN-1	0.831	0.832	0.864	0.802
NN-2	0.829	0.831	0.870	0.796
NN-3	0.827	0.824	0.834	0.814
NN-4	0.829	0.829	0.851	0.807
NN-5	0.831	0.832	0.866	0.802

Table 5.9: Results of testing trained roberta-base model with different neural networks as classifiers.

5.6 LIME Explanation

In order to use LIME explanation on top of the trained model, *LimeTextExplainer* class is imported from lime[10] library. Its *explain_instance()* function generates explanations for a prediction and requires 4 parameters:

1. **Data_row:** 1d numpy array corresponding to a row or a claim for which explanations want to be generated.

2. **Predict_fn:** Prediction function that takes an array of claims and outputs the prediction probabilities for each claim.

3. **Num_features:** Maximum number of features present in the explanation. For highlighting the importance of particular words, the number 10 was chosen as a sufficient choice.

4. **Num_samples:** Size of the neighbourhood to learn the linear model. This specifies the number of different samples that are created from the initial claim. The higher the number, the slower your prediction is generated. 1000 samples were chosen as a sufficient number to give meaningful explanations within a short time.

The function creates num_samples variations of an input claim. Then each claim goes through the classifier through *lime_predictor()* method (implementation of *Predict_fn()*), and the lime library calculates the influence words have on the outcome of the classification (as discussed in Section 2.7.2). The end results are presented visually. This explanation is necessary for the web application to enhance the trustworthiness of the model and provide the user with a meaningful output. The ability to generate LIME explanations was first implemented in the 'Models_Implementation.ipynb' notebook (Appendix C.3.1 Chapter 11) and then transformed to 'lime_explainer.py' file in the Streamlit website code (Appendix C.4.4).

[10] `https://lime-ml.readthedocs.io`

5.7 Streamlit Web Tool

Streamlit[11] was chosen for this project because it provides the quickest way to deploy machine learning models. It is an open-source Python library run on Flask that is used for creating web applications. It is easy to use, highly structured, focuses on simplicity and does not have any requirements with JavaScript or even CSS. Its UI is not as customisable as Flask[12] or Dash[13], which are other frameworks for deploying web apps. However, it provides enough options for implementing a simple application - a goal of this project.

The implemented web application called 'Fake Take', provides a way to predict the veracity of a given claim. After the claim is written and a button 'validate' is clicked, the application inputs that claim into a pre-loaded and already trained roberta-base model that outputs a prediction probability. In addition to the prediction itself, the LIME explanation is presented to the user after a short time to allow for more insight into the model's inner workings. The user is also given an option to see a short explanation of how the output is generated and what does it mean. This is all accessible from the 'Home' page as seen in Figure 5.4.

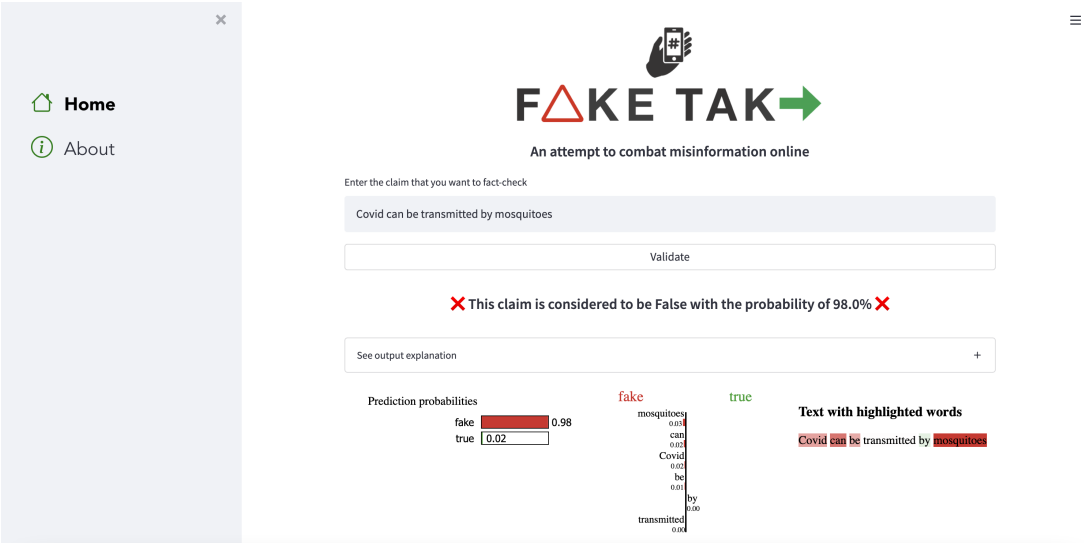


Figure 5.4: 'Home' page of the implemented website.

In addition to the main front page, an 'About' section can be opened through the navigation sidebar. This section offers more information about the web application, the project itself, its main intentions and explanations of the approach. It also offers quick access to download the project report and redirect to the GitHub repository of this project (see Figure 5.5).

[11] `https://streamlit.io`
[12] `https://flask.palletsprojects.com`
[13] `https://dash.plotly.com`

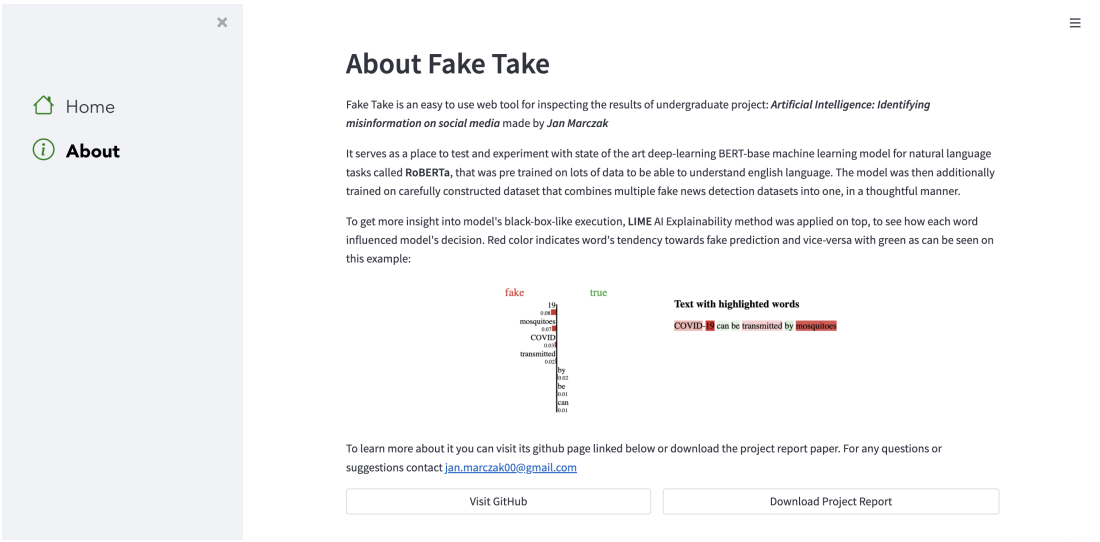


Figure 5.5: 'About' page of the implemented website.

Overall the main purpose of the app is to be able to display the results of this project. The initial goal was not to make the website public just yet, hence the code can for now be run only through the terminal window.

5.7.1 Files Structure

This section lays down the structure of the application files and explains how particular components were implemented. All of the code described in this section is placed in Appendix C.4.

app.py

App.py is the main file of the application that gets called at the beginning of each web session. It specifies the necessary configuration of the website (like its layout, sidebar style, CSS files etc.) and initialises its main components by calling the *sidebar()* method.

layout.py

This file defines the user interface of the website. It creates the main layout for both the 'Home' and 'About' pages. It is also responsible for displaying all the results outputted by the classifier, alongside its LIME explanations. Whenever the 'Validate' button is clicked, the *display_prediction_result()* and *display_lime_explanation()* functions are called that run the remaining files.

lime_explaier.py

lime_explainer.py is the file that contains the methods for generating and formatting LIME explanations after the prediction is made. Its main *lime_predictor()* function follows the same structure as the *Predict_fn* discussed in Section 5.6. Because of Streamlit's limitations with formatting some front-end components, an additional *format_lime_html()* is implemented to allow for changing colors associated with lime graphical explanation.

model.py

This file contains a direct copy of *BertForFakeNewsDetection()* class discussed in Section 5.4.3, however this time for RoBERTa model. Besides defining a model architecture and loading it from a file (roberta-base.pt), it also stores necessary data tokenization functions to be done after the claim is inputted to the website.

Chapter 6

Evaluation

This section evaluates different aspects of this project by looking both at the implementation and the bigger picture of automatic fake news detection. It also offers ideas on how particular components could be improved to overcome their shortcomings.

6.1 Data Quality

One of the main focuses of this project was attempting to eliminate bias and improve data quality for fake news detection. As already discussed in Section 2.6, publicly available datasets for this particular problem tend to have a lot of flaws and unusual characteristics that help machine learning classifiers in finding undesirable patterns. For example, in the Kaggle dataset fake-labelled claims are the only data entries that contain capitalised words. Hence, it is not a surprise that researchers achieve very high F1 scores that can even reach the 0.99 mark using Kaggle data. With such datasets, the problem is suddenly transformed from 'fake news detection' into 'patterns in text detection'. This is an example of an issue that this project aimed to tackle by eliminating such characteristics from datasets based on individual analysis.

Whereas the bias has been eliminated to some degree, and evident defects have been deleted, it is nearly impossible to detect all of them automatically. Different pre-processing techniques can only attempt to deal with this issue and cannot provide a profound examination. On the other hand, a comprehensive manual inspection of fake news entries can prove valuable; however, this time-consuming and tedious process cannot guaranteed success. Oftentimes the bias can be hidden from the human eye, which makes this problem even more difficult.

Implemented Google Search Verifier has been used to determine the veracity of the Twit-

terFakeNews dataset and half-true entries of LIAR. Although the results seem promising and useful, this solution is far from perfect. A human being should not base their knowledge on the main page of Google results without making sure that the results come exclusively from trustworthy websites. This could be improved by making sure that the scraped titles are only from specific domains. The Levenshtein distance metric also favours claims with the same length as the website titles. This was mitigated by adjusting the score based on the claim's length. However, something alongside Cosine similarity could be implemented to measure the correlation between the two texts. Such an approach usually yields better results but is also much slower than the Levenshtein distance. Ideally, none of the half-true entries or whole TwitterFakeNews dataset should have been used because of their noise and uncertainty. However, with this, the data size would decrease dramatically.

Nonetheless, the whole process of creating a dataset and verifying some of its claims proved more successful than picking just one available dataset. The F1 score of the build RoBERTa model is smaller in comparison to attempts from researchers using a single dataset. This was expected since the data is now less predictable to the machine learning classifiers. Merging several different datasets does not make the news dataset complete, however, it eliminates the possibility of the model picking up on specific unusual characteristics. If the fake news detection models with an accuracy of over 95%, were actually able to perform that well on unseen data from different sources, the problem of misinformation online would have been solved by now.

6.2 BERT Models Evaluation

This section focuses on interpreting the results of various BERT-based models presented in Table 5.7. The main takeaway is that RoBERTa models were the highest performing transformer-based classifiers for this particular task. Its improved training and fine-tuning procedures over the original BERT explain their superior results. Additionally, large models outperformed their equivalent smaller versions, which suggests that a higher number of layers increases the model's abilities. Unsurprisingly, cased models that recognise capital letters were also an improvement over uncased versions. One of the strange observations is that roberta-large training time was around twice as fast as the bert-large models. This is even more surprising, considering that roberta-base trained for slightly longer than bert-base moments. It is worth noting that in most models Recall was the metric that had the lowest score. This suggests that the models are having a harder time classifying true stories, and are overly cautious. Whereas an argument could be made that such wary classification is better, this is an aspect that could be looked

into.

The additional adjustment of the neural network's hyperparameters proved to be less successful. A simple ANN seems to have little to no influence over a complex multi-layered transformer-based model and serves only as a way to transform it into a classifier. Additional experimentation can be carry out by implementing and evaluating CNN and RNN on top of BERT word-embeddings. This might improve the model's performance but some of the discussed research papers suggest otherwise.

Overall, testing and evaluation were fair and sufficient, however, rushed due to time and resource limitations. In an ideal scenario, each model would have its hyperparameters adjusted individually. For example number of epochs was determined with the 'bert-base' model and used across all other training procedures. Instead, this could have been done for all models. Additionally, a higher number of hyperparameters could have been adjusted in order to maximise the potential of these classifiers. However, due to Google Colab's limitations in GPU usage, large-scale training and tuning would have been very challenging and time-consuming.

6.3 Fake News Detection Based on Word-Embeddings

The idea behind using BERT-based word embeddings for detecting misinformation online was to put a bigger emphasis on the actual text, rather than on other features often used for classifying news. The goal was to make the model understand the English language and be able to deduct the trustworthiness of claims related to any topic.

This has proven to be very challenging. RoBERTa model seems to struggle with distinguishing between various differently structured sentences. Even if it classifies something correctly, it only takes a minor adjustment to the sentence to make it shift its opinion about the claim. On the other hand, negated correctly classified claims often yield the same prediction, suggesting that the initial correct guess was down to pure luck. This has been surprising and undesirable behaviour. Table 6.1 shows some examples of sentences that display this unsatisfactory performance. The classification is green when the model predicted its label correctly, and red otherwise.

In hindsight, for this approach to work successfully, an enormous dataset would have had to be used that would cover primary facts and historical events that happened all around the world across hundreds of years. That task seems impossible to accomplish. Although this project's implemented model achieves a high F1 score on a thoroughly-constructed dataset, it cannot process the vast amount of new or historic news. It is however unwise to expect the

Claim	Label	Classifiaction	Probability
Barack Obama was the president of the United States of America	True	True	74%
Barack Obama was the president of USA	True	False	85%
Jan Marczak was the president of the United States of America	False	True	56%
Apple's new Iphone 14 will not be invisible	True	True	98%
Apple's new Iphone 14 will be invisible	False	True	89%
Russian troops are bombing Ukrainian cities	True	False	71%
Russian troops are not bombing Ukrainian cities	False	False	94%
England will face USA in the FIFA football world cup	True	True	100%
England will not face USA in the FIFA football world cup	False	True	99%

Table 6.1: Example of similarly structured claims and their results given by roberta-base model.

model to correctly classify the news it has not seen before. Hence, for such a technique to be effective the dataset would have to be updated each day and describe all main stories that are happening around the globe.

Another way to improve such a classifier would be to focus on one specific topic. This way the main objective would be to construct a reliable dataset related to a particular event like for example Coronavirus pandemic. This also presents its own issues, as the facts evolve and change over time, however by narrowing the scope of the model, its behaviour becomes less irregular.

6.4 LIME Effectiveness

LIME explanations implemented on top of the RoBERTa model serve as a way to investigate the prediction given by the classifier. Its clarifications cannot be explicitly used to determine exactly why a given claim was labelled in such a way. However, they provide an interesting insight into the model's behaviour. They are particularly useful when comparing very similar sentences with slight differences. For example, Figure 6.1 shows the explanations generated when inputting two correctly classified sentences:

"Barack Obama was the president of the United States" and

"Barack Obama was **not** the president of the United States"

It is clear that negating this sentence with the word 'not' changes the sentiment of the claim. The influence that it has on the model outweighs any other word present in this sentence and shifts it to predict false.

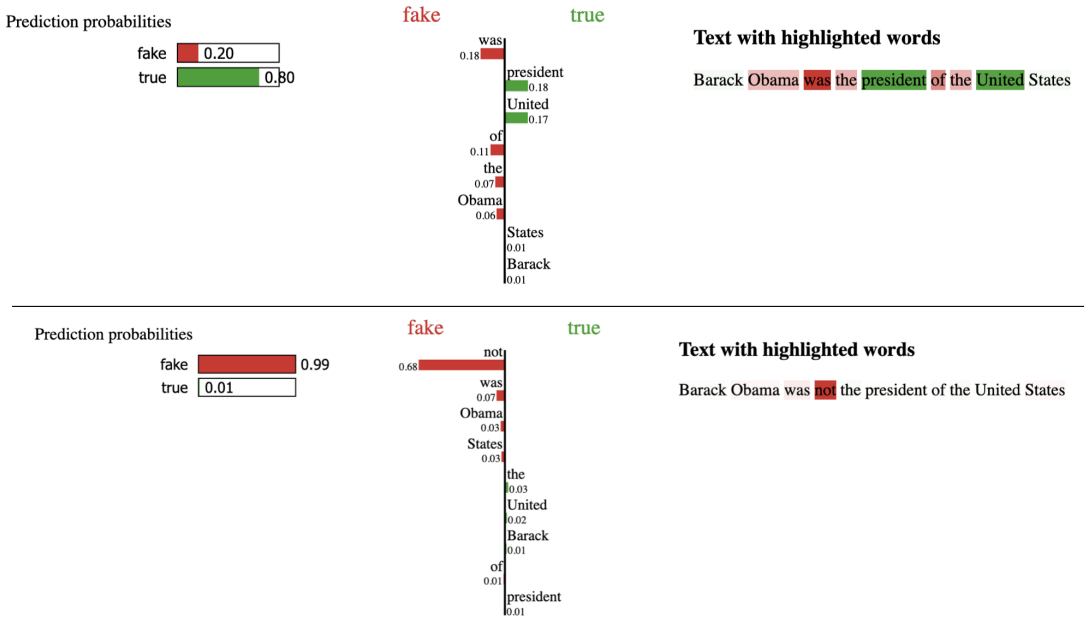


Figure 6.1: Examples of graphical LIME explanation provided for two very similarly structured sentences.

Although, in this case, this is the desired behaviour resulting in a sensible explanation, it is not always the case. Sometimes the influence words have on the prediction are minimal and do not offer any insight. The meaning of the numbers associated with each word is also unclear. This is especially confusing in examples like in Figure 6.2, where none of the words stands out, and the prediction is hard to grasp. Overall, LIME helps decide whether the model's prediction is trustworthy, but it should not be used instinctively.

6.5 Software Evaluation

A web application for inspecting results is the end product of this project. Because this work is far from a typical software engineering project, the website was not implemented with any particular methodology in mind. The main goal was to programme it quickly and present a user with a minimal, intuitive and pleasant looking interface and correct functionality. This was hopefully achieved as the website managed to be implemented accordingly to an early UI

Figure 6.2: Example of graphical LIME explanation that does not offer much insight into model's decision.

prototype drafted at the beginning of the project. Given the convenience of coding in Streamlit, the app has been structured accordingly to usual software engineering techniques. Each major component has been given its file to encapsulate the logic behind them.

Because the web application has not been deployed on any server, the implementation phase assumed that users interested in running this website will not be malicious agents. This means that the software is not necessarily well tested in terms of potential bugs, however since the functionality is somewhat straightforward, manual testing was conducted to ensure everything works correctly. The model itself accepts as input any type of claim, which may result in undesirable answers. However, once again the website assumes that users are here to validate checkable claims. To summarize, in its current state the app is meant for people that would like to use it, not to break it. In the future, the website could be deployed and necessary measures would be implemented to ensure its reliability.

Chapter 7

Legal, Social, Ethical and Professional Issues

Throughout the development of this project, The Code of Conduct laid out by the British Computing Society (BCS) has been followed at all times. Not many guidelines described in the document are directly relevant to the project, however great care has been carried out to make sure this work adheres to the principles of BCS.

The project's purpose is to fight the spread of misinformation online, which in itself is a very relevant issue affecting public well-being each day. To make sure users are aware of the risks and trustworthiness of the software, a disclaimer has been added to inform them that automated models should not be the main source of fake news verification. This is visible on the home page and is there to remember that this website is just an inspection tool, not a flawless fake news detector.

During the implementation part of the project, great care has been taken to ensure that any code used from external sources is appropriately cited. The code is my own work unless stated otherwise and some of its inspirations are also explicitly declared. Any open-source libraries have also been referenced to ensure the credit is given.

Lastly, the data used in this project has not been individually collected. Instead, publicly available datasets for fake news detection have been utilised. At the beginning of the project both ethical clearance and King's Data Protection Registration have been considered, however, due to the nature of the project and lack of personal data collection, no further action was required.

Chapter 8

Conlusion

This project aimed to provide an automated way to verify information in a more thorough way than what had been done previously. It focused on combining all the necessary components for attempting a fake news classifier: constructing a diversified dataset, experimenting with best performing models, adding explainable AI and providing accessible software.

In its way it has been rather successful, accomplishing all listed elements and implementing this comprehensive approach for fake news detection. The dataset construction eliminated a lot of flawed data entries and their undesired characteristics. The best performing model achieved an F1 Score of 85%, LIME explainability added a level of trustworthiness and a website provided easy access to the project's components.

Nevertheless, relying explicitly on word embeddings and a single dataset (albeit unbiased) to produce accurate predictions about the veracity of any claims proved to be ineffective. Misinformation turned out to be too broad of a subject to solve with one machine learning model. News evolves over time, facts change, hundreds of new storylines appear in the world each day and automated classifiers learned on limited data cannot comprehend these dynamic aspects.

More research and work need to be done to combat this growing problem. This project could serve as one component in a more complicated automated fake news detector that combines multiple methods of classifying information. Additionally, explainable AI is also still an immature field and more explainability methods need to be developed in order to fully comprehend deep learning models.

Chapter 9

Future Work

To improve the capabilities of this project in automatically identifying misinformation, many improvements and additions could be implemented in future work.

To offset the limitations of utilising solely word embeddings, ensemble learning techniques could be used to combine multiple algorithms to obtain better predictive performance. Each of these models could focus on different types of features. One model could look at the features extracted from texts, whereas the other could focus on social media users and their activity (if the analysed text was posted on a particular network). In order to keep up with the sheer amount of new information appearing each day, an additional model that checks and analyses trustworthy news websites could also be implemented. This way the most mainstream news could be fact-checked using websites like *BBC* or *CNN*. Assuming that these models would be based on neural network architectures they could be combined with a single neural network classifier.

As stated in Chapter 8, to reduce the complexity of the problem and focus on effectiveness rather than completeness, a more topic-oriented fake news classifier could be implemented. It could focus on COVID-19 related news or the horrifying war in Ukraine - an even more relevant subject that is used to spread false propaganda online. This way the model has a bigger chance of success and can be later improved to cover more news.

Additionally, the website could also be improved by adding more features and insights, but most importantly, it could be deployed and hosted on some platform. Explainable AI techniques are still an immature technology and more research needs to be done in order to improve Deep Learning XAI methods.

References

[1] M. Walker and K. E. Matsa, "News consumption across social media in 2021," Sep 2021.

[2] "Fighting misinformation in the time of covid-19, one click at a time."

[3] Y. Roth and N. Pickles, "Updating our approach to misleading information," 2020.

[4] Instagram, "Combatting misinformation on instagram," 2019.

[5] Q. Wong, "More harm than good? twitter struggles to label misleading covid-19 tweets," May 2020.

[6] D. Lee, "Instagram is hiding faked images, and it could hurt digital artists," Jan 2020.

[7] A. Woodward, "'fake news': A guide to trump's favourite phrase – and the dangers it obscures," Oct 2020.

[8] S. Helmstetter and H. Paulheim, "Collecting a large scale dataset for classifying fake news tweets using weak supervision," *Future Internet*, vol. 13, no. 5, p. 114, 2021.

[9] J. C. Reis, A. Correia, F. Murai, A. Veloso, and F. Benevenuto, "Explainable machine learning for fake news detection," in *Proceedings of the 10th ACM conference on web science*, pp. 17–26, 2019.

[10] A. Nguyen, K. Pham, D. Ngo, T. Ngo, and L. Pham, "An analysis of state-of-the-art activation functions for supervised deep neural network," in *2021 International Conference on System Science and Engineering (ICSSE)*, pp. 215–220, IEEE, 2021.

[11] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[13] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of machine learning research*, vol. 12, no. 7, 2011.

[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[15] Y. Yang, L. Zheng, J. Zhang, Q. Cui, Z. Li, and P. S. Yu, "Ti-cnn: Convolutional neural networks for fake news detection," *arXiv preprint arXiv:1806.00749*, 2018.

[16] "Fake news."

[17] P. Bahad, P. Saxena, and R. Kamal, "Fake news detection using bi-directional lstm-recurrent neural network," *Procedia Computer Science*, vol. 165, pp. 74–82, 2019.

[18] U. Isha Priyavamtha, G. Vishnu Vardhan Reddy, P. Devisri, and A. S. Manek, "Fake news detection using artificial neural network algorithm," in *International Conference on Information Processing*, pp. 328–340, Springer, 2021.

[19] N. Chaibi, B. Atmani, and M. Mokaddem, "Deep learning approaches to intrusion detection: A new performance of ann and rnn on nsl-kdd," in *Proceedings of the 1st International Conference on Intelligent Systems and Pattern Recognition*, pp. 45–49, 2020.

[20] Y. Li and T. Yang, "Word embedding for understanding natural language: a survey," in *Guide to big data applications*, pp. 83–104, Springer, 2018.

[21] K. Li, "Haha at fakedes 2021: A fake news detection method based on tfidf and ensemble machine learning," in *Proceedings of the Iberian Languages Evaluation Forum (IberLEF 2021)*, 2021.

[22] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[25] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the 2018 Conference of the*

North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), Association for Computational Linguistics, June 2018.

[26] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[27] A. Rogers, O. Kovaleva, and A. Rumshisky, "A primer in bertology: What we know about how bert works," *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 842–866, 2020.

[28] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[29] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

[30] D. Q. Nguyen, T. Vu, and A. T. Nguyen, "Bertweet: A pre-trained language model for english tweets," *arXiv preprint arXiv:2005.10200*, 2020.

[31] H. Jwa, D. Oh, K. Park, J. M. Kang, and H. Lim, "exbake: Automatic fake news detection model based on bidirectional encoder representations from transformers (bert)," *Applied Sciences*, vol. 9, no. 19, p. 4062, 2019.

[32] R. K. Kaliyar, A. Goswami, and P. Narang, "Fakebert: Fake news detection in social media with a bert-based deep learning approach," *Multimedia tools and applications*, vol. 80, no. 8, pp. 11765–11788, 2021.

[33] D. Kar, M. Bhardwaj, S. Samanta, and A. P. Azad, "No rumours please! a multi-indic-lingual approach for covid fake-tweet detection," *arXiv preprint arXiv:2010.06906*, 2020.

[34] N. R. de Oliveira, P. S. Pisa, M. A. Lopez, D. S. V. de Medeiros, and D. M. Mattos, "Identifying fake news on social networks based on natural language processing: Trends and challenges," *Information*, vol. 12, no. 1, p. 38, 2021.

[35] P. Patwa, S. Sharma, S. Pykl, V. Guptha, G. Kumari, M. S. Akhtar, A. Ekbal, A. Das, and T. Chakraborty, "Fighting an infodemic: Covid-19 fake news dataset," *arXiv preprint arXiv:2011.03327*, 2020.

[36] T. Mitra and E. Gilbert, "Credbank: A large-scale social media corpus with associated credibility annotations," in *Ninth international AAAI conference on web and social media*, 2015.

[37] G. C. Santia and J. R. Williams, "Buzzface: A news veracity dataset with facebook user commentary and egos," in *Twelfth International AAAI Conference on Web and Social Media*, 2018.

[38] K. Shu, D. Mahudeswaran, S. Wang, D. Lee, and H. Liu, "Fakenewsnet: A data repository with news content, social context and spatialtemporal information for studying fake news on social media," *arXiv preprint arXiv:1809.01286*, 2018.

[39] W. Y. Wang, "" liar, liar pants on fire": A new benchmark dataset for fake news detection," *arXiv preprint arXiv:1705.00648*, 2017.

[40] J. Thorne, A. Vlachos, C. Christodoulopoulos, and A. Mittal, "Fever: a large-scale dataset for fact extraction and verification," *arXiv preprint arXiv:1803.05355*, 2018.

[41] L. Cui and D. Lee, "Coaid: Covid-19 healthcare misinformation dataset," *arXiv preprint arXiv:2006.00885*, 2020.

[42] A. Zubiaga, M. Liakata, R. Procter, G. Wong Sak Hoi, and P. Tolmie, "Analysing how people orient to and spread rumours in social media by looking at conversational threads," *PloS one*, vol. 11, no. 3, p. e0150989, 2016.

[43] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st international conference on neural information processing systems*, pp. 4768–4777, 2017.

[44] C. Molnar, "Interpretable machine learning," Mar 2022.

[45] S. Knapič, A. Malhi, R. Saluja, and K. Främling, "Explainable artificial intelligence for human decision support system in the medical domain," *Machine Learning and Knowledge Extraction*, vol. 3, no. 3, pp. 740–770, 2021.

[46] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1135–1144, 2016.

[47] E. Kokalj, B. Škrlj, N. Lavrač, S. Pollak, and M. Robnik-Šikonja, "Bert meets shapley: Extending shap explanations to transformer-based classifiers," in *Proceedings of the EACL Hackashop on News Media Content Analysis and Automated Report Generation*, pp. 16–21, 2021.

[48] X. Man and E. P. Chan, "The best way to select features? comparing mda, lime, and shap," *The Journal of Financial Data Science*, vol. 3, no. 1, pp. 127–139, 2021.

[49] O. M. Adam Thorne, Zac Farnsworth, "Research of lstm additions on top of squad bert hidden transform layers," *Stanford, CA 94305*, no. 1, p. 9.

[50] M. Popel and O. Bojar, "Training tips for the transformer model," *arXiv preprint arXiv:1804.00247*, 2018.

Contents

Appendix A

Full Results of BERT-based Models Testing

All of the calculated metrics, confusion matrices and ROC curves related to each BERT-based model and different neural networks classifiers are presented in this appendix.

A.1 Different BERT-based Models With Initial Neural Network

This section presents testing results of different BERT-based models with initial neural network. The initial neural network (which turned out to be best performing neural network) that the experimentation was conducted has an architecture presented in Table A.1.

Dropout Layer	Dropout(0.2)
Hidden Layer	Linear(BERT_output, 50)
Activation Function 1	ReLU()
Output Layer	Linear(50, 2)
Activation Function 2	LogSoftmax

Table A.1

The following are the results of different BERT-based models with this neural network:

A.1.1 bert-base-uncased

Accuracy	Recall	Precision	F1 Score	Training Time	Model Size
0.796	0.805	0.781	0.793	27 min	438 mb

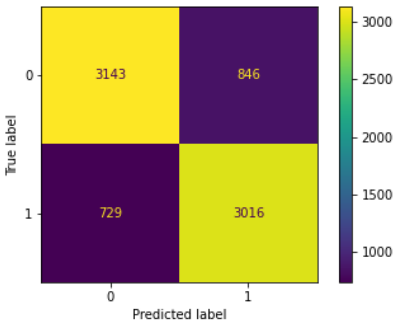


Figure A.1: bert-base-uncased confusion matrix

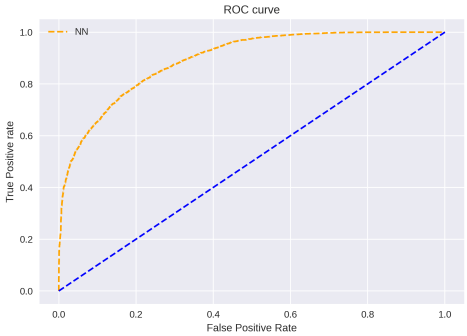


Figure A.2: bert-base-uncased ROC curve

A.1.2 bert-base-cased

Accuracy	Recall	Precision	F1 Score	Training Time	Model Size
0.824	0.813	0.821	0.817	28 min	433 mb

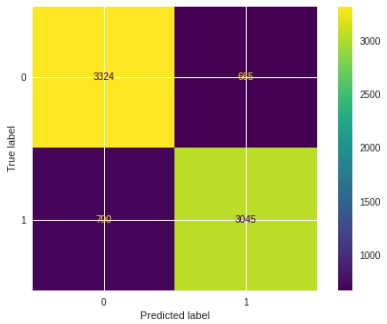


Figure A.3: bert-base-cased confusion matrix

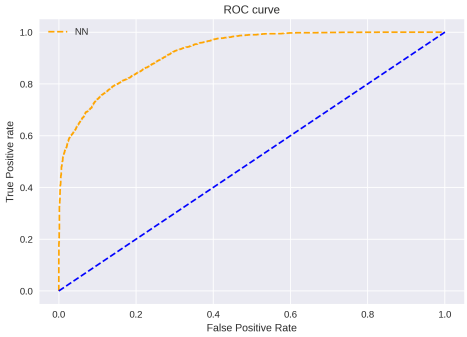


Figure A.4: bert-base-cased ROC curve

A.1.3 bert-large-uncased

Accuracy	Recall	Precision	F1 Score	Training Time	Model Size
0.803	0.833	0.777	0.804	89 min	1.34 gb

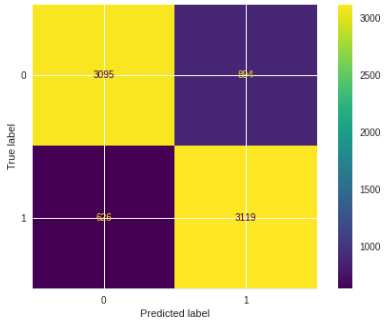


Figure A.5: bert-large-uncased confusion matrix

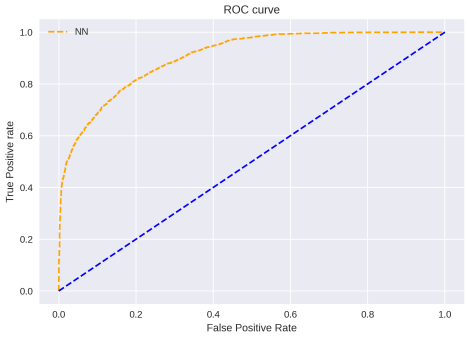


Figure A.6: bert-large-uncased ROC curve

A.1.4 bert-large-cased

Accuracy	Recall	Precision	F1 Score	Training Time	Model Size
0.825	0.828	0.814	0.821	87 min	1.34 gb

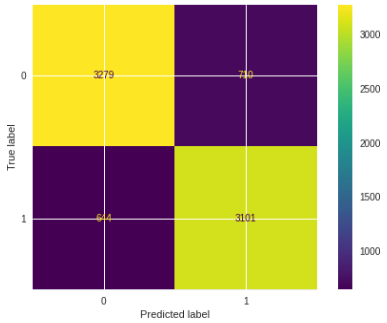


Figure A.7: bert-large-cased confusion matrix

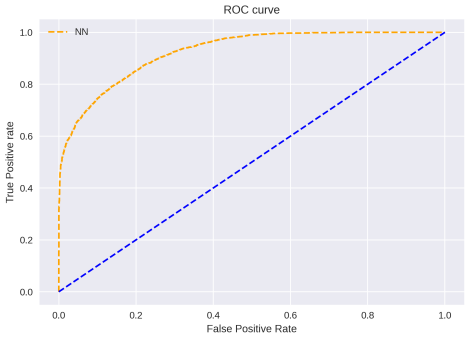


Figure A.8: bert-large-cased ROC curve

A.1.5 roberta-base

Accuracy	Recall	Precision	F1 Score	Training Time	Model Size
0.831	0.864	0.802	0.832	30 min	500 mb

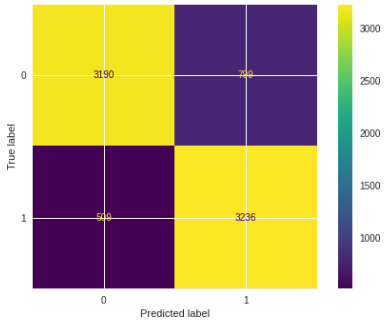


Figure A.9: roberta-base confusion matrix

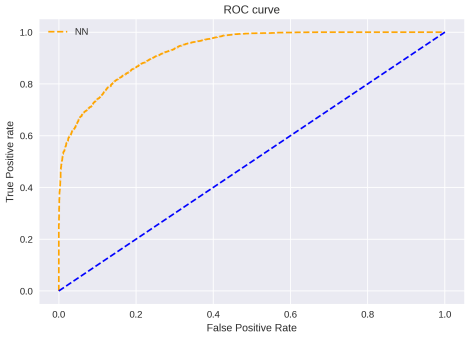


Figure A.10: roberta-base ROC curve

A.1.6 roberta-large

Accuracy	Recall	Precision	F1 Score	Training Time	Model Size
0.840	0.864	0.816	0.839	46 min	1.42 gb

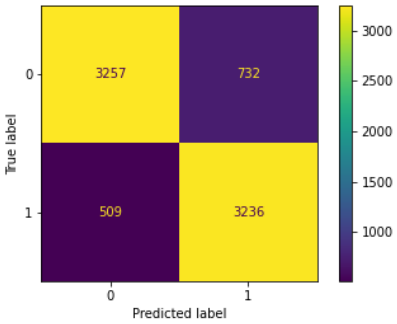


Figure A.11: roberta-large confusion matrix

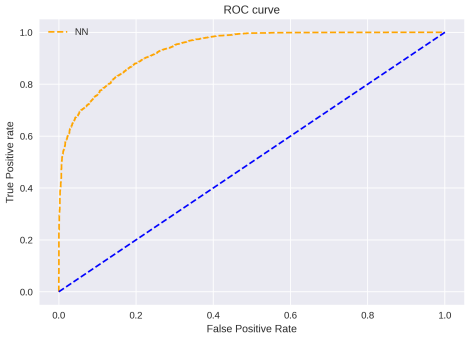


Figure A.12: roberta-large ROC curve

A.1.7 bertweet-base

Accuracy	Recall	Precision	F1 Score	Training Time	Model Size
0.834	0.853	0.813	0.832	15 min	540 mb

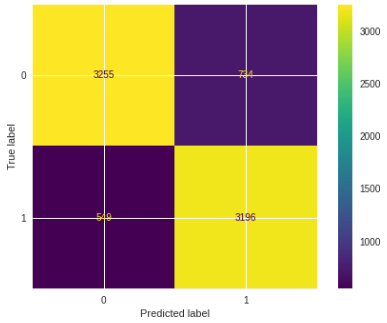


Figure A.13: bertweet-base confusion matrix

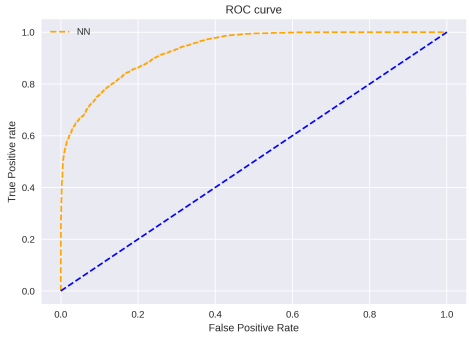


Figure A.14: bertweet-base ROC curve

A.1.8 albert-base-v2

Accuracy	Recall	Precision	F1 Score	Training Time	Model Size
0.792	0.767	0.797	0.782	27 min	47 mb

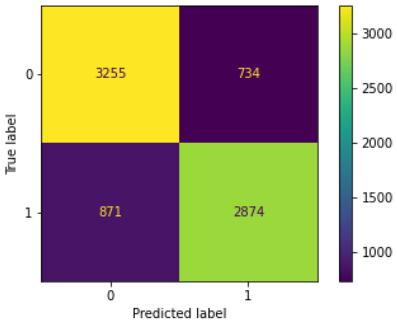


Figure A.15: albert-base-v2 confusion matrix

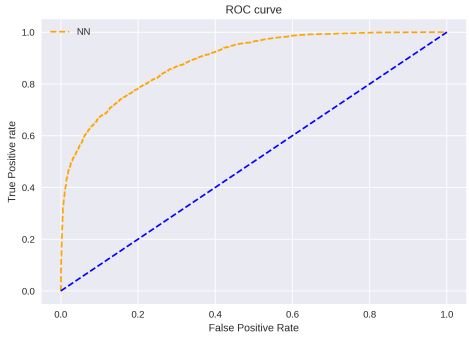


Figure A.16: albert-base-v2 ROC curve

A.2 Different Neural Networks With roberta-base Model

This section presents results of different neural network classifiers with roberta-base model.

A.2.1 Neural Network 1

Dropout Layer	Dropout(0.2)
Hidden Layer	Linear(BERT_output, 50)
Activation Function 1	ReLU()
Output Layer	Linear(50, 2)
Activation Function 2	LogSoftmax

Table A.2: neural network 1 classifier architecture

Accuracy	**Recall**	**Precision**	**F1 Score**
0.831	0.864	0.802	0.832

Table A.3: neural network 1 results

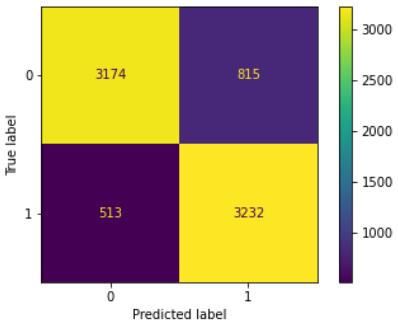


Figure A.17: neural network 1 confusion matrix

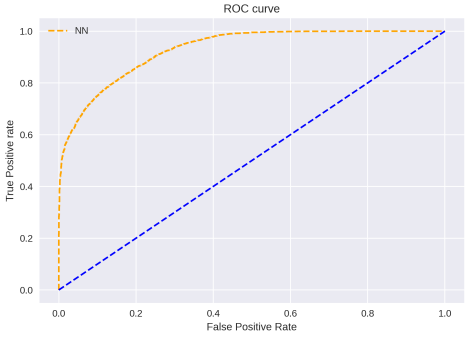


Figure A.18: neural network 1 ROC curve

A.2.2 Neural Network 2

Dropout Layer	Dropout(0.2)
Hidden Layer	Linear(BERT_output, 50)
Activation Function 1	LeakyReLU()
Output Layer	Linear(50, 2)
Activation Function 2	LogSoftmax

Table A.4: neural network 2 classifier architecture

Accuracy	**Recall**	**Precision**	**F1 Score**
0.829	0.870	0.796	0.831

Table A.5: neural network 2 results

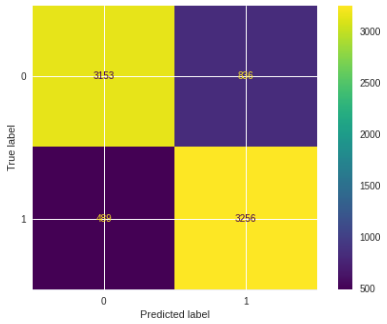


Figure A.19: neural network 2 confusion matrix

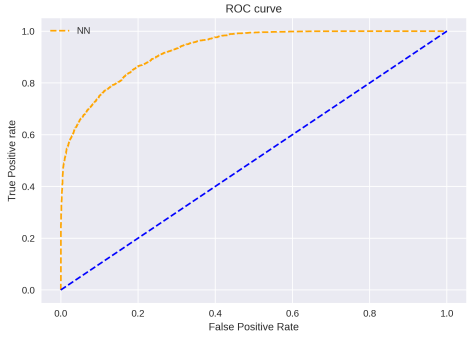


Figure A.20: neural network 2 ROC curve

A.2.3 Neural Network 3

Dropout Layer	Dropout(0.2)
Hidden Layer	Linear(BERT_output, 256)
Activation Function 1	LeakyReLU()
Output Layer	Linear(256, 2)
Activation Function 2	Softmax

Table A.6: neural network 3 classifier architecture

Accuracy	**Recall**	**Precision**	**F1 Score**
0.827	0.834	0.814	0.824

Table A.7: neural network 3 results

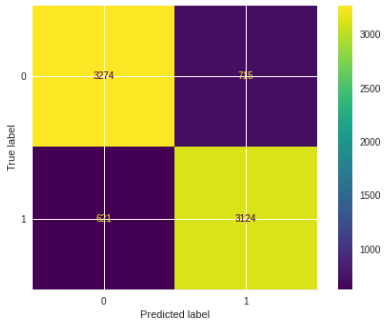


Figure A.21: neural network 3 confusion matrix

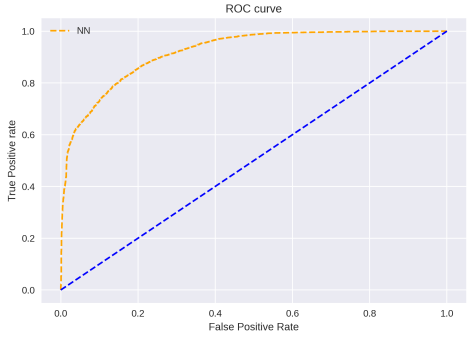


Figure A.22: neural network 3 ROC curve

A.2.4 Neural Network 4

Dropout Layer	Dropout(0.2)
Hidden Layer	Linear(BERT_output, 50)
Activation Function 1	SELU()
Output Layer	Linear(50, 2)
Activation Function 2	LogSoftmax

Table A.8: neural network 4 classifier architecture

Accuracy	**Recall**	**Precision**	**F1 Score**
0.829	0.851	0.807	0.829

Table A.9: neural network 4 results

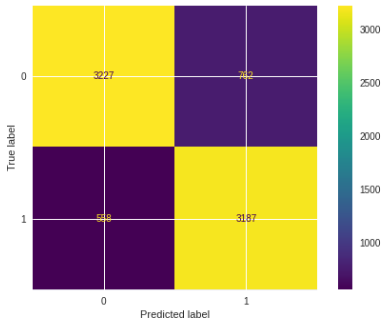


Figure A.23: neural network 4 confusion matrix

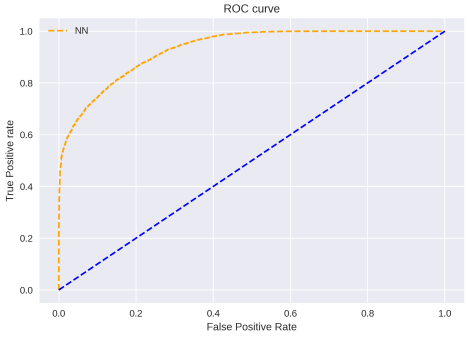


Figure A.24: neural network 4 ROC curve

A.2.5 Neural Network 5

Dropout Layer	Dropout(0.2)
Hidden Layer	Linear(BERT_output, 50)
Activation Function 1	GELU()
Output Layer	Linear(50, 2)
Activation Function 2	Logoftmax

Table A.10: neural network 5 classifier architecture

Accuracy	**Recall**	**Precision**	**F1 Score**
0.831	0.866	0.802	0.833

Table A.11: neural network 5 results

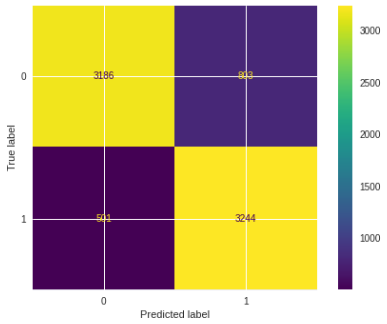


Figure A.25: neural network 5 confusion matrix

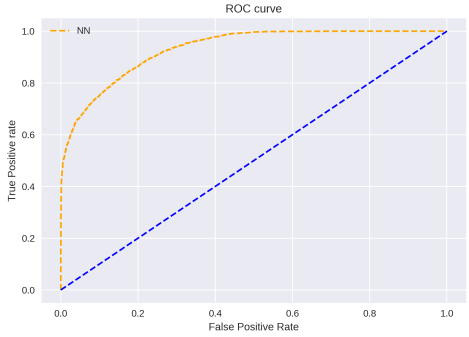


Figure A.26: neural network 5 ROC curve

Appendix B

User Guide

Because the project is far from a Software Engineering project, most of the implemented components do not require a user guide. All of the code for Datasets Analysis, Models Implementation and Google Search Verifier can be run like every other Jupyter Notebook code. Users should follow the *README.md* file instruction submitted with the project.

In regards to the implemented website the software is very simple and most of its components are visible on the initial home screen. The user can input any claim he wants to see model's prediction on its veracity and additional explainability. Apart from this, an 'About' page can be accessed through the side menu, where the user can learn more about the website and the whole project.

Appendix C

Program Listings

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary - Jan Marczak, 8/04/2022

Program Listings contain all source code implemented in this project. This includes 4 main sections:

- C.1 Google Search Verifier Script - Code implementing Google Search Verification script that was used to calculate Leventshein Distance between queries and the front-page Google results.
- C.2 Dataset Analysis and Construction Notebooks - Seven Jupyter Notebooks containing analysis of particular datasets and construction of this project's dataset in the last section.
- C.3 Models Implementation and Evaluation - Big Jupyter Notebook containing all the code responsible for creating, fine-tuning, evaluating and experimenting with different BERT-based models.
- C.4 Streamlit Website Code - Source code for the website that showcases the deliverables of this project.

To allow for easier inspection and readability subsections in this appendix are named in the same way as actual submission files.

Unfortunately, due to an unknown issue with Plotly, some graphs in Jupyter Notebooks had to be deleted. They can still be viewed in original code but printing the file as pdf caused their deletion.

C.1 Google Search Verifier Script

C.1.1 Google_Search_Verifier.ipynb

GOOGLE SEARCH VERIFIER

Google Search Verifier, given the sentence forms a query and scrapes the results from the front page of google. It then calculates Levenshtein score that measures the similarity between the initial query and the results. This is done to help with unreliable data.

Necessery Imports

In [1]:
```
import pandas as pd
import numpy as np
import urllib.request
import ssl
import random
import time
import json
from leven import levenshtein
from bs4 import BeautifulSoup
ssl._create_default_https_context = ssl._create_unverified_context
```

List of User Agents

- Google makes it hard to scrape the results from their search engine. In order to prevent google from banning a "bot that scrapes its results" user-agents have to be definined to trick google into thinking that human is behind the computer
- In each query a random user-agent is picked from this list

In [2]:
```
user_agents = [
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 S
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.56 (KHTML, like Gecko) Chrome/87.0.4283.88 S
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_5) AppleWebKit/537.61 (KHTML, like Gecko) Chrome/87.0.4284.88 S
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_17_5) AppleWebKit/532.60 (KHTML, like Gecko) Chrome/87.0.4288.88 S
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_17_3) AppleWebKit/532.90 (KHTML, like Gecko) Chrome/87.0.4281.88 S
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5) AppleWebKit/537.74.9 (KHTML, like Gecko) Chrome/87.0.4281.88
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.2526.106
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.2526.106
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.163
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.96 S
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3946.132
]
```

Google Scraper

The algorithm works as follows:

- Get random user agents lists and a dataset of claims
- For each row in dataset
  - create a google query
  - wait 5 seconds
  - Create a request (add random user agent)
  - Read the response
  - Parse the results
  - Initialise score and results to 0
  - For each div that appear on the main page:
    - Get the title
    - Calculate Levenstein score between title and the query
    - Save leven_score and increment results
  - Save overall leven_score divided by nr of the results scraped

In [6]:
```
# dataframe -> dataframe that needs an attribute named "claim" that will be queried in google
# waittime -> time that the algorithm waits before executing 2 queries in a row. Default = 5s
def google_scrape(dataframe, waittime = 5):
    for i, row in dataframe.iterrows():

        # Get the claim and form a google query
        claim = row.tweet__text
#         claim = row.claim
        query = claim.replace(" ", "+")
        query = 'https://google.com/search?q=' + query

        # wait "waittime" seconds in between each query
        time.sleep(waittime)

        # Send the request and parse results
        request = urllib.request.Request(query)
        request.add_header('User-agent', random.choice(user_agents))
        raw_response = urllib.request.urlopen(request).read()
        html = raw_response.decode("utf-8")
        soup = BeautifulSoup(html, 'html.parser')
        divs = soup.select("#search div.g")

        results = 0
        leven_score = 0
        # Scrape the titles that appear on the main page
        for div in divs:
            result = div.select("h3")
            if (len(result) >= 1):
                title = result[0].get_text()
                if title[-3:] == '...':
                    title = title[:-4]

                # Calculate levenshtein score between the initial claim and a google result (title)
                leven_score += levenshtein(claim, title)
                results += 1

            if results != 0:
                dataframe.at[i,'leven__score'] = leven_score/results
            else:
                dataframe.at[i,'leven__score'] = -1 # omething went wrong put
```

In [5]:
```
def getResults(dataframe, data_name):
    google_scrape(dataframe, 4)
    print(dataframe.head(5))
    dataframe.to_csv(data_name, encoding='utf-8')
```

Performing Google Scraping on particular datasets

1. Test

Running code below would take ages, hence a cell to test the script is provided below

In [8]:
```
data = [['Beyonce faces $20M copyright suit from Youtube stars estate', 1, 0],
        ['Site is getting blown up now thanks to Real Clear Politics', 0, 0]]
df = pd.DataFrame(data, columns = ['tweet__text', 'claim_veracity', 'leven__score'])

getResults(df, 'results.csv')
```

```
                                      tweet__text  claim_veracity  \
0  Beyonce faces $20M copyright suit from Youtube...               1
1  Site is getting blown up now thanks to Real Cl...               0

   leven__score
0           5.3
1          46.6
```

2. TwitterFakeNews

2.1 Fake entries

In [5]:
```
Auto_df_fake = pd.read_csv('../Datasets_analysis/TwitterFakeNews/Auto_Format_Fake.csv',
                           sep=';', usecols = ['tweet__fake', 'tweet__text'], low_memory=False)

# Add additional column leven_score
Auto_df_fake['leven__score'] = 0.0
getResults(Auto_df_fake, 'Auto_df_fake_score.csv')
```

2.2 True Entries

In [7]:
```
Auto_df_true = pd.read_csv('../Datasets_analysis/TwitterFakeNews/Auto_Format_True.csv', sep=';',
                           usecols = ['tweet__fake', 'tweet__text'], low_memory=False)

# Add additional column leven_score
Auto_df_true['leven__score'] = 0.0
getResults(Auto_df_true, 'Auto_df_true_score.csv')
```

3. LIAR Half-True entries

In [ ]:
```
LIAR_half_true_df = pd.read_csv('../Datasets_analysis/LIAR/LIAR_half_true.csv',
                                usecols = ['claim_veracity', 'claim'], low_memory=False)

# Add additional column leven_score
LIAR_half_true_df['leven__score'] = 0.0
getResults(LIAR_half_true_df, 'LIAR_half_true_df.csv')
```

C.2 Datasets Analysis and Construction Notebooks

C.2.1 LIAR.ipynb

LIAR Dataset

12k+ entries of authentic, real-world short statements from various contexts with diverse speakers and topics

https://paperswithcode.com/dataset/liar

Neccessery imports

In [1]:
```
import pandas as pd
import numpy as np
import plotly.express as px
import plotly
plotly.offline.init_notebook_mode(connected=True)

import warnings
warnings.filterwarnings('ignore')
```

1. Loading the training, testing and validation datasets and merging into one

In [3]:
```
LIAR_train_df = pd.read_csv('Initial_datasets/train.csv', delimiter='\t', header=None,
                            index_col=False, low_memory=False)
LIAR_test_df = pd.read_csv('Initial_datasets/test.csv', delimiter='\t', header=None,
                           index_col=False, low_memory=False)
LIAR_valid_df = pd.read_csv('Initial_datasets/valid.csv', header=None, index_col=False, low_memory=False)

LIAR_df = pd.concat([LIAR_train_df, LIAR_test_df, LIAR_valid_df], ignore_index=True)
LIAR_df.set_axis(['json', 'claim_veracity', 'claim', 'topics', 'speaker', 'job', 'state',
                  'political_party', 'credit_history_1', 'credit_history_2', 'credit_history_3',
                  'credit_history_4', 'credit_history_5', 'context'], axis=1, inplace=True)
print("size od df: ", LIAR_df.shape)
LIAR_df.head(2)
```

```
size od df:  (12791, 14)
```

Out[3]:

	json	claim_veracity	claim	topics	speaker	job	state	political_party	credit_history_1	credit_history_2
0	2635.json	false	Says the Annies List political group supports ...	abortion	dwayne-bohac	State representative	Texas	republican	0.0	1.0
1	10540.json	half-true	When did the decline of coal start? It started...	energy,history,job-accomplishments	scott-surovell	State delegate	Virginia	democrat	0.0	0.0

2. Doping useless columns

Dropping credit history as it's very specific for this dataset and won't be useful in further analysis

In [4]:
```
LIAR_df.drop(['json', 'political_party', 'job', 'state', 'credit_history_1', 'credit_history_2',
              'credit_history_3', 'credit_history_4', 'credit_history_5'], axis=1, inplace=True)
LIAR_df.head(5)
```

Out[4]:

	claim_veracity	claim	topics	speaker	context
0	false	Says the Annies List political group supports ...	abortion	dwayne-bohac	a mailer
1	half-true	When did the decline of coal start? It started...	energy,history,job-accomplishments	scott-surovell	a floor speech.
2	mostly-true	Hillary Clinton agrees with John McCain "by vo...	foreign-policy	barack-obama	Denver
3	false	Health care reform legislation is likely to ma...	health-care	blog-posting	a news release
4	half-true	The economic turnaround started at the end of ...	economy,jobs	charlie-crist	an interview on CNN

3. Columns formating

3.1 Topics

Changing topics to be an array of topics not a string

In [5]:
```
LIAR_df['topics'] = LIAR_df['topics'].str.split(',')
```

3.2 Speaker

ex: from hilary-clinton to Hilary Clinton

In [6]:
```
LIAR_df['speaker'] = LIAR_df['speaker'].str.replace("-", " ").str.title()
```

3.3 Claims

- There are many claims in this dataset that paraphrase what speaker said ex: "Says John McCain has done nothing to help the vets."
- Because I am more focused on the actual claim rather than who said it, whenever I can I want to eliminate the speaker ex: "John McCain has done nothing to help the vets."
- Sometimes however if the claim is personal I add the speaker before the claim to form a sentence such as: "Donald Trump says John McCain has done nothing to help the vets."

In [9]:
```
# Sentences that have his/him/he/hes/she/shes/her/hers will have a speaker added to them
personal_list = (' his ', ' him ', ' he ', ' hes ', ' she ', ' shes ', ' her ', ' hers ')
pattern = '|'.join(personal_list)
LIAR_df.loc[((LIAR_df['claim'].str.startswith('Says')) & LIAR_df['claim'].str.contains(pattern)),
            'claim'] = LIAR_df['speaker'].str[:] + " " + LIAR_df['claim'].str[:]

# Deleting Says that
LIAR_df.loc[LIAR_df['claim'].str.startswith('Says that'), "claim"] = LIAR_df['claim'].str[10:]

# Deleting Says a
LIAR_df.loc[LIAR_df['claim'].str.startswith('Says a '), "claim"] = LIAR_df['claim'].str[7:]

# Deleting Says the
LIAR_df.loc[LIAR_df['claim'].str.startswith('Says the '), "claim"] = LIAR_df['claim'].str[5:]

# Deleting Says the
LIAR_df.loc[LIAR_df['claim'].str.startswith('Says '), "claim"] = LIAR_df['claim'].str[5:]

# Deleting clais that start with "On " and don't contain ':' -> They usually don't have any claim
# Example claim: 'On mandating health care coverage' - It doesn't tell us anything
LIAR_df.drop(LIAR_df.loc[(LIAR_df['claim'].str.lower().str.startswith('on ')) &
                         (LIAR_df['claim'].str.contains(':') == False)].index, inplace=True)

# Adding the speaker to the remaining claims that start with 'On '
LIAR_df.loc[LIAR_df['claim'].str.startswith('On '), 'claim'] = LIAR_df['speaker'].str[:] + " o" + LIAR_df['claim']

# Capitalise the claims
LIAR_df['claim'] = LIAR_df['claim'].str[0].str.capitalize() + LIAR_df['claim'].str[1:]
```

In [11]:
```
# LIAR_df[(LIAR_df['claim'].str.lower().str.startswith('on ')) & (LIAR_df['claim'].str.len() > 40)]
LIAR_df[(LIAR_df['claim'].str.lower().str.contains('on')) & (LIAR_df['claim'].str.contains(':'))].head(5)
```

Out[11]:

	claim_veracity	claim	topics	speaker	context
149	barely-true	Donald Trump on the VA: Over 300,000 veterans ...	[health-care, veterans]	Donald Trump	a speech.
188	mostly-true	It is a commitment voters take very seriously:...	[elections, taxes]	Americans Tax Reform	a news release
267	half-true	Hypocrisy at the Clinton Foundation: Top male ...	[candidates-biography, women, workers]	Donald Trump	an Instagram post
357	true	You know we can't just pull out now... The tru...	[iraq]	Joe Biden	CNN/YouTube debate in Charleston, S.C.
375	pants-fire	Sheila Jackson Lee of Texas said: Hey, all you...	[candidates-biography, diversity, campaign-adv...	Facebook Posts	a meme supposedly quoting Sheila Jackson Lee

3.4 claim_veracity

- TRUE: true, mostly-true
- FALSE: false, pants-fire, barely-true
- half-true data entries will go through google search verifier

In [ ]:
```
fig = px.histogram(LIAR_df, x='claim_veracity').update_xaxes(categoryarray=['pants-fire', 'false', 'barely-true',
fig.update_layout(bargap=0.2)
fig.show()
```

In [13]:
```
conditions = [LIAR_df['claim_veracity'].eq('true'),
              LIAR_df['claim_veracity'].eq('mostly-true'),
              LIAR_df['claim_veracity'].eq('TRUE'),
              LIAR_df['claim_veracity'].eq('pants-fire'),
              LIAR_df['claim_veracity'].eq('false'),
              LIAR_df['claim_veracity'].eq('barely-true'),
              LIAR_df['claim_veracity'].eq('FALSE')]
choices = [True, True, True, False, False, False, False]
LIAR_df['claim_veracity'] = np.select(conditions, choices, default = LIAR_df['claim_veracity'])
```

Half-true claims will be a grey area of this dataset as they will be the hardest to categorise and requires the most amount of time to investigate. It's also the biggest category so adding all to either "false" or "true" will shift the balance quite dramatically.

In [ ]:
```
fig = px.histogram(LIAR_df, x='claim_veracity').update_xaxes(categoryarray=['pants-fire', 'false', 'barely-true',
fig.update_layout(bargap=0.2)
fig.show()
```

4. Inspecting characteristics

4.1 Topics distribution

In [ ]:
```
# Replacing NaN with empty arrays
isna = LIAR_df['topics'].isna()
LIAR_df.loc[isna, 'topics'] = pd.Series([[]] * isna.sum()).values

topics_occurances = LIAR_df.topics.sum()
topics_dict = {i:topics_occurances.count(i) for i in set(topics_occurances)}

fig = px.pie(LIAR_df, values=list(topics_dict.values()), names=list(topics_dict.keys()))
fig.update_traces(textposition='inside', textinfo='value')
fig.show()
```

We can clearly see that this dataset is diverse and doesn't focus on one specific topic

4.1 Context

Context modification

The context was very specfic so this changes the context to more be more general

In [14]:
```
print(len(LIAR_df['context'].unique())) #Over 5k possible contexts...

LIAR_df['context'] = LIAR_df['context'].fillna('None')

# Array that define change in context given certain keywords
speech_context = [['speech', 'senate', 'state', 'house', 'floor', 'meeting',
                   'answer', 'commentary', 'presentation', 'hearing', 'opinion',
                   'talk', 'appearance', 'interview', 'debate', 'radio', 'rally',
                   'remark', 'conference', 'discussion', 'comment', 'town', 'response'], 'speech']
add_context = [['ad', 'mailer', 'flier', 'billboard', 'commercial', 'flyer', 'brochure', 'campaign'], 'ad']
social_media_context = [['twitter', 'tweet', 'facebook', 'post', 'social media',
                         'media', 'web', 'forum', 'blog', 'internet', 'video', 'message'], 'social media']
news_context = [['press', 'news', 'website', 'cnn', 'tv', 'release', 'article', 'segment',
             'abc', 'nbc', 'television', 'fox', 'msnbc', 'editiorial', 'column', 'op-ed'], 'news']
writing_context = [['book', 'email', 'e-mail', 'letter', 'report', 'autobiography', 'report', 'petition', 'survey'

# Changing context given keywords
contexts_array = [speech_context, add_context, social_media_context, news_context, writing_context]
for context in contexts_array:
    context_regex = "|".join(context[0])
    LIAR_df.loc[LIAR_df['context'].str.lower().str.contains(context_regex), 'context'] = context[1]


# Other contexts are treated as undefined (they are very vague)
possible_contexts = ['speech', 'ad', 'social media', 'news', 'writing']
possible_contexts_regex = "|".join(possible_contexts)
LIAR_df.loc[~LIAR_df['context'].str.lower().str.contains(possible_contexts_regex), 'context'] = 'Undefined'
```

```
5047
```

New Context Distribution

In [ ]:
```
fig = px.pie(LIAR_df, names = 'context', color = 'context', color_discrete_sequence=px.colors.qualitative.Pastel)
fig.update_traces(text = LIAR_df['context'].value_counts(), textinfo = 'label+percent')
fig.show()
```

5. Examining Google-verified Half-True entries

Half-true entries were fed into google_scraper, where the average levensthein distance between the claim and the google query results was calculated. This allows to more in-depth analysis and observations on whereas a half-true claim should be considered true or false

In [15]:
```
LIAR_df = LIAR_df[LIAR_df['claim_veracity'] != 'half-true']
```

5.1. Loading the dataset

In [17]:
```
pd.set_option("display.max_rows", None, "display.max_columns", None)
pd.options.display.max_colwidth = 200

LIAR_score = pd.read_csv('../../Google_Scraper/LIAR_verified/LIAR_half_true_score.csv',
                         index_col=0, low_memory=False)

# Delete the entries with leven_score = 0
LIAR_score = LIAR_score.drop(LIAR_score.loc[LIAR_score['leven__score'] <= 0].index)
```

In [18]:
```
# Deleting entries that were omitted before
LIAR_score.drop(LIAR_score.loc[(LIAR_score['claim'].str.lower().str.startswith('on')) &
                               (LIAR_score['claim'].str.contains(':') == False)].index, inplace=True)
```

5.2. Adjusting levenshtein distance score

Adjusting leven_score to account for the claim character count

In [19]:
```
LIAR_score['claim_count'] = LIAR_score['claim'].str.len()
LIAR_score['adjusted_leven'] = LIAR_score['leven__score'] / LIAR_score['claim_count']
```

5.3. Splitting into groups based on sentence length

From each group the 50% of half-true entries that had the best (smallest) adjusted_leven will be considered as true and false vice-versa

In [20]:
```
print("4 different ranges of characters length that split the dataset into 4 groups: \n\n",
      pd.cut(LIAR_score['claim_count'], 4).unique())

LIAR_score['char_category'] = pd.cut(LIAR_score['claim_count'], 4, labels=[0, 1, 2, 3])
LIAR_score.sort_values('adjusted_leven').head(5)
```

```
4 different ranges of characters length that split the dataset into 4 groups:

 [(94.75, 170.5], (18.697, 94.75], (170.5, 246.25], (246.25, 322.0]]
Categories (4, interval[float64]): [(18.697, 94.75] < (94.75, 170.5] < (170.5, 246.25] < (246.25, 322.0]]
```

Out[20]:

	claim_veracity	claim	leven__score	claim_count	adjusted_leven	char_category
1805	half-true	Harvard Study Finds States With Most Gun Laws Have Fewest Gun Deaths.	37.222222	69	0.539452	0
1436	half-true	People (are) paying more in taxes than they will for food, housing and clothing combined.	48.272727	89	0.542390	0
988	half-true	Americans will spend more on taxes in 2015 than on food, clothing and housing combined.	47.333333	87	0.544061	0
812	half-true	Undocumented immigrantspay $12 billion a year into Social Security.	37.400000	67	0.558209	0
2029	half-true	Americans will spend more on taxes in 2014 than they will on food, clothing and housing combined.	56.545455	97	0.582943	1

5.4. Taking 50% of best/worse entries from each group to be true/false

In [21]:
```
dfs=[]

for category in [0,1,2,3]:
    category_df = LIAR_score[LIAR_score['char_category'] == category].sort_values('adjusted_leven')
    half_index = int(len(category_df) * 0.5)

    true_df = category_df.iloc[:half_index]
    true_df['claim_veracity'] = True
    dfs.append(true_df)

    false_df = category_df.iloc[half_index:]
    false_df['claim_veracity'] = False
    dfs.append(false_df)

half_true_df = pd.concat(dfs)
LIAR_Final = pd.concat([half_true_df, LIAR_df])
```

In [22]:
```
LIAR_Final.head(2)
```

Out[22]:

	claim_veracity	claim	leven__score	claim_count	adjusted_leven	char_category	topics	speaker	context
1805	True	Harvard Study Finds States With Most Gun Laws Have Fewest Gun Deaths.	37.222222	69.0	0.539452	0	NaN	NaN	NaN
1436	True	People (are) paying more in taxes than they will for food, housing and clothing combined.	48.272727	89.0	0.542390	0	NaN	NaN	NaN

6. Saving modified dataset

In [ ]:
```
# Dropping unncessery columns, renamining and rearranging remaining one
LIAR_Final = LIAR_Final.sample(frac=1).reset_index(drop=True).drop(['leven__score', 'claim_count', 'adjusted_leven
LIAR_Final = LIAR_Final[['claim', 'claim_veracity']]
LIAR_Final['claim_veracity'] = LIAR_Final['claim_veracity'].astype(int)

LIAR_Final.to_csv('LIAR_Final.csv')
```

C.2.2 FEVER.ipynb

FEVER

150k short factual statements from Wikipedia annotated by trained personel, with 3 levels of claim veracity: supported, disprovided and not enough information

Necessery imports

In [1]:
```
import pandas as pd
import numpy as np
import plotly.express as px
import plotly
import json
pd.set_option('display.max_rows', 500, "display.max_colwidth", None)
plotly.offline.init_notebook_mode(connected=True)

import plotly.offline as pyo
import plotly.graph_objs as go
pyo.init_notebook_mode()
```

1. Loading the Dataset

In [3]:
```
FEVER_df = pd.read_json('Initial_datasets/FEVER_initial.jsonl', lines=True)
FEVER_df.shape[0]
```

Out[3]: 145449

2. Examining the dataset

2.1 Dropping unnessery columns and rows

- We don't need the id or the evidence so we can drop these columns

In [4]:
```
FEVER_df = FEVER_df.drop(['id', 'evidence'], 1)
```

- We also don't need the rows with "Not enough info" as we want clear indiciation if a claim is true or false. This also means that the "Not verifiable" rows will be deleted as there is a one-to-one relationship between these two.

In [5]:
```
FEVER_df = FEVER_df.drop(FEVER_df.loc[FEVER_df['label'].str.lower().str.contains('not enough info')].index)
FEVER_df.loc[FEVER_df['verifiable'].str.lower().str.contains('not verifiable')].shape[0]

# Every claim is now "verifiable" so we can drop this column too
FEVER_df = FEVER_df.drop(['verifiable'], 1)
```

2.2 Examining true/fake entries

In [6]:
```
# Change the type of the entry to boolean with 1 meaning TRUE and 0 meaning it's a FALSE claim
conditions = [FEVER_df['label'].eq('SUPPORTS'),
              FEVER_df['label'].eq('REFUTES')]
choices = [1, 0]
FEVER_df['claim_veracity'] = np.select(conditions, choices, default = FEVER_df['label'])
FEVER_df = FEVER_df.drop(['label'], 1)
```

In [ ]:
```
fig = px.histogram(FEVER_df, x='claim_veracity').update_xaxes(categoryarray=[1, 0])
fig.update_layout(bargap=0.2)
fig.show()
```

2.3 Removing duplicates

In [8]:
```
# Removind duplicates (over 7k)
print(FEVER_df.shape[0])
FEVER_df = FEVER_df.drop_duplicates(subset='claim', keep='first')
FEVER_df.shape[0]
```

Out[8]:
```
109810
102292
```

2.4 Claim Length

- Group by sentence length and take some % of each group into the dataset

In [ ]:
```
FEVER_df['claim_count'] = FEVER_df['claim'].str.len()

# Showing claim length distribution
fig = px.histogram(FEVER_df, x='claim_count')
fig.update_layout(bargap=0.2)
fig.show()
```

In [10]:
```
print("8 different ranges of characters length that split the dataset into 8 groups: \n\n",
      pd.qcut(FEVER_df['claim_count'], 8).unique())

FEVER_df['char_category'] = pd.qcut(FEVER_df['claim_count'], 8, labels=[0, 1, 2, 3, 4, 5, 6, 7])
```

```
8 different ranges of characters length that split the dataset into 8 groups:

 [(57.0, 68.0], (29.0, 35.0], (68.0, 614.0], (50.0, 57.0], (35.0, 40.0], (40.0, 45.0], (45.0, 50.0], (10.999, 29.0
]]
Categories (8, interval[float64]): [(10.999, 29.0] < (29.0, 35.0] < (35.0, 40.0] < (40.0, 45.0] < (45.0, 50.0] < (
50.0, 57.0] < (57.0, 68.0] < (68.0, 614.0]]
```

In [11]:
```
# 500 random entries from each category will be chosen

dfs = []
for category in [0, 1, 2, 3, 4, 5, 6, 7]:
    category_df_false = FEVER_df[(FEVER_df['char_category'] == category) & (FEVER_df['claim_veracity'] == 0)]
    category_df_true = FEVER_df[(FEVER_df['char_category'] == category) & (FEVER_df['claim_veracity'] == 1)]
    dfs.append(category_df_true.sample(250, random_state=1))
    dfs.append(category_df_false.sample(250, random_state=1))

FEVER_picked = pd.concat(dfs)
```

3. Saving new datasets

3.1. Whole FEVER

In [87]:
```
FEVER_df = FEVER_df.sample(frac=1).reset_index(drop=True).drop(['claim_count', 'char_category'], axis=1)
FEVER_df.to_csv('FEVER_Final.csv', encoding='utf-8')
```

3.2. Picked FEVER

In [88]:
```
FEVER_picked = FEVER_picked.sample(frac=1).reset_index(drop=True).drop(['claim_count', 'char_category'], axis=1)
FEVER_picked.to_csv('FEVER_Picked.csv', encoding='utf-8')
```

C.2.3 CovidFakeNews.ipynb

COVID-19 News Dataset

10k+ social media posts and articles on COVID-19 manually labelled as true or false by a team

https://arxiv.org/abs/2011.03327

Necessery Imports

In [1]:
```
import pandas as pd
import numpy as np
import plotly.express as px
import plotly
pd.set_option('display.max_rows', 500, "display.max_colwidth", None)
plotly.offline.init_notebook_mode(connected=True)
```

1. Loading the datasets and merging them into one

In [2]:
```
COVID_df_1 = pd.read_csv('Initial_datasets/COVID_train.csv', usecols = ['tweet', 'label'], low_memory=False)
COVID_df_2 = pd.read_csv('Initial_datasets/english_test_with_labels.csv', usecols = ['tweet', 'label'], low_memory
COVID_df_3 = pd.read_csv('Initial_datasets/Constraint_val.csv', usecols = ['tweet', 'label'], low_memory=False)

COVID_df = pd.concat([COVID_df_1, COVID_df_2, COVID_df_3], ignore_index=True)
print("size od dataset: ", COVID_df.shape)
COVID_df.head()
```

```
size od dataset:  (10700, 2)
```

Out[2]:

	tweet	label
0	The CDC currently reports 99031 deaths. In general the discrepancies in death counts between different sources are small and explicable. The death toll stands at roughly 100000 people today.	real
1	States reported 1121 deaths a small rise from last Tuesday. Southern states reported 640 of those deaths. https://t.co/YASGRTT4ux	real
2	Politically Correct Woman (Almost) Uses Pandemic as Excuse Not to Reuse Plastic Bag https://t.co/thF8GuNFPe #coronavirus #nashville	fake
3	#IndiaFightsCorona: We have 1524 #COVID testing laboratories in India and as on 25th August 2020 36827520 tests have been done : @ProfBhargava DG @ICMRDELHI #StaySafe #IndiaWillWin https://t.co/Yh3ZxknnhZ	real
4	Populous states can generate large case counts but if you look at the new cases per million today 9 smaller states are showing more cases per million than California or Texas: AL AR ID KS KY LA MS NV and SC. https://t.co/1pYW6cWRaS	real

2. Examining the dataset

In [ ]:
```
fig = px.histogram(COVID_df, x='label').update_xaxes(categoryarray=['real', 'fake'])
fig.update_layout(bargap=0.2)
fig.show()
```

2.1 URLs, Hashtags and Mentions

Delete Links, Hashtags and @ (mentions) at the end of tweets:

Many entries include not only the claim we are interested in but also links to some resources, mentions and hashtags. Big portions of those come at the end of the tweet so by deleting them, the claim is cleaned from unncessery characters

In [5]:
```
COVID_df['modified_tweet'] = COVID_df['tweet']

# Keep deleting the last word of a tweet if it's a link, hashtag (#) or a mention (@).
# This usually does not influence the claim
stop = False
while stop == False:
    prev_tweets = COVID_df['modified_tweet'].copy(deep=True)
    COVID_df['modified_tweet'] = np.where((COVID_df['modified_tweet'].str.lower().str.rsplit(' ', 1).str[1].str.co
                                           (COVID_df['modified_tweet'].str.lower().str.rsplit(' ', 1).str[1].str.con
                                           (COVID_df['modified_tweet'].str.lower().str.rsplit(' ', 1).str[1].str.con
                                           COVID_df['modified_tweet'].str.rsplit(' ', 1).str[0], COVID_df['modified_

    if prev_tweets.equals(COVID_df['modified_tweet']):
        stop = True
```

Delete remaining entries with mentions (@) and link

These entries have mentions and links in the middle of the tweet, therefore there wouldn't be useful when training our model, as we assume the input won't include any arbitrary text

In [6]:
```
# link in the middle or at the front: 167, delete
COVID_df = COVID_df.drop(COVID_df.loc[COVID_df['modified_tweet'].str.lower().str.contains('http') == True].index)

# mention (@) still in the modified_tweet: 1443, delete - hard to encode
COVID_df = COVID_df.drop(COVID_df.loc[COVID_df['modified_tweet'].str.lower().str.contains('@') == True].index)
```

Examining remaining hashtags

These entries have mentions and links in the middle of the tweet, therefore there wouldn't be useful when training our model, as we assume the input won't include any arbitrary text

In [7]:
```
# Replacing the most frequent #COVID19 hashtag to normal word before further analysis and formatting
COVID_df['modified_tweet'] = COVID_df['modified_tweet'].str.replace('#COVID19 ','COVID-19 ')
COVID_df['modified_tweet'] = COVID_df['modified_tweet'].str.replace('#COVID-19 ','COVID-19 ')
COVID_df['modified_tweet'] = COVID_df['modified_tweet'].str.replace('#COVID_19 ','COVID-19 ')
COVID_df['modified_tweet'] = COVID_df['modified_tweet'].str.replace('#COVID ','COVID-19 ')
```

In [8]:
```
# Drop the entries that include hashtags in the form of "CoronaVirusUpdate" i.e.
# entries that have hashtags with multiple words chained together (around 800 rows)
COVID_df = COVID_df.drop(COVID_df.loc[(COVID_df['modified_tweet'].str.contains(r'#[A-Z][a-z]+[A-Z]'))].index)
```

In [9]:
```
# Dropping entries that have #COVID19 and other stuff attatched to it like #COVIDNigeria
COVID_df = COVID_df.drop(COVID_df.loc[COVID_df['modified_tweet'].str.lower().str.contains('#covid19[^ ,?.-]')].ind
```

In [10]:
```
# Dropping entries that have 1 or more hashtags at the beginning of the claim (80)
COVID_df = COVID_df.drop(COVID_df.loc[COVID_df['modified_tweet'].str.match(r'#[a-zA-Z\d]+ #*')].index)
```

In [11]:
```
# Entries with hashtag and ":" 35 -> Drop the first word
COVID_df['modified_tweet'] = np.where(COVID_df['modified_tweet'].str.match('#[a-zA-Z\d]+:'),
                                      COVID_df['modified_tweet'].str.split(' ', 1).str[1],
                                      COVID_df['modified_tweet'])
```

In [12]:
```
# Still 522 entries have # in them (middle of the sentence) -> Drop the #
COVID_df['modified_tweet'] = COVID_df['modified_tweet'].str.replace('#','')
```

2.2 Removing Duplicates

In [13]:
```
# Removind duplicates (around 120)
COVID_df = COVID_df.drop_duplicates(subset='modified_tweet', keep='first')
```

2.3 Other Special Characters

&

In [14]:
```
# Entries containing '&amp' (300) -> Drop
COVID_df = COVID_df.drop(COVID_df[COVID_df['modified_tweet'].str.contains('&amp')].index)
```

\n

In [15]:
```
# Tweets containing '\n' (180) -> Drop
COVID_df = COVID_df.drop(COVID_df[COVID_df['modified_tweet'].str.contains('\n')].index)
```

&

In [16]:
```
# Tweets containing ' & ' -> Replace with ' and ' instead (25)
COVID_df['modified_tweet'] = COVID_df['modified_tweet'].str.replace(' & ',' and ')
# Drop the tweets that have no spacing around '&' (the rest)
COVID_df = COVID_df.drop(COVID_df[COVID_df['modified_tweet'].str.contains('&')].index)
```

;

In [17]:
```
# Tweets containing '; [a-z]' (42) -> Switch to '. [a-z]'
COVID_df['modified_tweet'] = np.where(COVID_df['modified_tweet'].str.contains(r'; [a-z]+'),
                                      COVID_df['modified_tweet'].str.replace('; ',', '),
                                      COVID_df['modified_tweet'])


# Tweets containing '; [A-Z]' (17) -> Switch to '. [A-Z]'
COVID_df['modified_tweet'] = np.where(COVID_df['modified_tweet'].str.contains(r'; [A-Z]+'),
                                      COVID_df['modified_tweet'].str.replace('; ','. '),
                                      COVID_df['modified_tweet'])

# Rest of the tweets containing ';' (mostly gibbrish) -> Drop
COVID_df = COVID_df.drop(COVID_df[COVID_df['modified_tweet'].str.contains(';')].index)
```

Last sentence check

In [18]:
```
# After out prevoious formatting we need to look at the end of the tweets

# Tweets that end with : (500) -> Deleting the last sentence
COVID_df['modified_tweet'] = COVID_df['modified_tweet'].str.replace(r"(?<=[.!?])[^.!?]*?:\s*$", "", regex=True)
```

Non-ascii characters

In [19]:
```
# Deleting non-ascii characters from the strings
COVID_df['modified_tweet'] = COVID_df['modified_tweet'].astype(str).apply(lambda x: x.encode('ascii', 'ignore').de
```

First character check

In [20]:
```
# SENTENCE wirth unusual starting character (340):
# - multiple starting with _
# - multiple starting with ?
COVID_df.loc[COVID_df['modified_tweet'].str.match(r'[^a-zA-Z0-9"$]')].shape[0]

COVID_df['modified_tweet'] = np.where(COVID_df['modified_tweet'].str.startswith('_'),
                                      COVID_df['modified_tweet'].str.replace('_',''),
                                      COVID_df['modified_tweet'])

COVID_df['modified_tweet'] = np.where(COVID_df['modified_tweet'].str.startswith('?'),
                                      COVID_df['modified_tweet'].str.replace(r'[?]+','', regex=True),
                                      COVID_df['modified_tweet'])
```

???

In [21]:
```
# 166 entries with '???' -> delete the ???
COVID_df[COVID_df['modified_tweet'].str.lower().str.contains('??', regex=False)].head(100)

COVID_df['modified_tweet'] = np.where(COVID_df['modified_tweet'].str.contains('???', regex=False),
                                      COVID_df['modified_tweet'].str.replace('???', '', regex=False),
                                      COVID_df['modified_tweet'])
```

2.4 Sentence length

There are two main reasons to use logarithmic scales in charts and graphs. The first is to respond to skewness towards large values; i.e., cases in which one or a few points are much larger than the bulk of the data. The second is to show percent change or multiplicative factors.

In [ ]:
```
COVID_df['tweet_count'] = COVID_df['modified_tweet'].str.len()

fig = px.histogram(COVID_df, x='tweet_count') # With log scale to see a better distribution
fig.update_layout(bargap=0.2)
fig.show()

# 6000 entries with modified_tweet character count less than 200
# COVID_df = COVID_df[COVID_df['modified_tweet'].str.len() < 200]
```

2.5 True/Fake distribution

In [ ]:
```
fig = px.histogram(COVID_df, x='label').update_xaxes(categoryarray=['real', 'fake'])
fig.update_layout(bargap=0.2)
fig.show()

# Change the type to boolean type
COVID_df['label'] = (COVID_df['label'] == 'real').astype(int)
```

3. Saving the dataset

In [24]:
```
COVID_df_final = COVID_df[['modified_tweet', 'label']]
COVID_df_final.rename(columns = {'modified_tweet': 'claim', 'label': 'claim_veracity'}, inplace=True)
```

In [25]:
```
COVID_df_final.to_csv('CovidFakeNews_Final.csv', encoding='utf-8')
```

C.2.4 CoAID.ipynb

CoAID: COVID-19 Healthcare Misinformation Dataset

CoAID (Covid-19 heAlthcare mIsinformation Dataset) is a diverse COVID-19 healthcare misinformation dataset, including fake news on websites and social platforms, along with users' social engagement about such news. It includes 5,216 news, 296,752 related user engagements, 958 social platform posts about COVID-19, and ground truth labels.

This dataset is taken from here: https://github.com/cuilimeng/CoAID

Necessery Imports

In [1]:
```
import pandas as pd
import numpy as np
# import plotly.express as px
# import plotly
pd.set_option('display.max_rows', 500, "display.max_colwidth", None)
# plotly.offline.init_notebook_mode(connected=True)
```

1. Loading the Datasets

In [2]:
```
CoAID_claim_fake = pd.read_csv('Initial_datasets/ClaimFakeCOVID.csv',
                               usecols = ['title'], index_col = False, low_memory=False)
CoAID_claim_real = pd.read_csv('Initial_datasets/ClaimRealCOVID.csv',
                               usecols = ['title'], index_col = False, low_memory=False)
CoAID_news_fake = pd.read_csv('Initial_datasets/NewsFakeCOVID.csv',
                              usecols = ['title'], index_col = False, low_memory=False)
CoAID_news_real = pd.read_csv('Initial_datasets/NewsRealCOVID.csv',
                              usecols = ['title'], index_col = False, low_memory=False)
```

In [3]:
```
# Adding claim_veracity column
CoAID_claim_real['claim_veracity'] = 1
CoAID_claim_fake['claim_veracity'] = 0
CoAID_news_real['claim_veracity'] = 1
CoAID_news_fake['claim_veracity'] = 0

# Merging df in pairs
CoAID_claim = pd.concat([CoAID_claim_real, CoAID_claim_fake], ignore_index=True)
CoAID_news = pd.concat([CoAID_news_real, CoAID_news_fake], ignore_index=True)

# Merging all dfs
CoAID = pd.concat([CoAID_claim, CoAID_news], ignore_index=True)
```

In [4]:
```
CoAID.shape[0]
```

Out[4]:
```
5975
```

2. Formatting the Dataset

2.1. Deleting quotes around claims

Multiple claims are wrapped around qoutation marks, which is unnecessary

In [4]:
```
def deleteQuotes(df):
    df['title'] = np.where((df['title'].str[0] == '"') & (df['title'].str[-1] == '"'),
                           df['title'].str[1:-1], df['title'])
    df['title'] = np.where((df['title'].str[0] == '”') & (df['title'].str[-1] == '”'),
                           df['title'].str[1:-1], df['title'])

for df in [CoAID_claim, CoAID_news]: deleteQuotes(df)
```

2.2. Deleting Weird characters from the dataset

In [5]:
```
# Deleting non-ascii characters from the strings

def deleteNonAscii(df):
    df['title'] = df['title'].astype(str).apply(lambda x: x.encode('ascii', 'ignore').decode('ascii'))

for df in [CoAID_claim, CoAID_news]: deleteNonAscii(df)
```

2.3. Deleting entries with Links

In [6]:
```
def deleteLinks(df):
    df.drop(df.loc[df['title'].str.lower().str.contains('http')].index, inplace=True)

for df in [CoAID_claim, CoAID_news]: deleteLinks(df)
```

2.3. Inspecting CoAID_claim dataset (518 entries)

- Claims as questions : Many claims are questions themselves and can't really be classified as true or false. They are all labelled as true, however they can't really be classified so need to be dropped as faulty entries

In [7]:
```
CoAID_claim_picked = CoAID_claim[CoAID_claim['title'].str[-1] != '?'] # We are left only with 75 entries
```

- Other entries are of good quality and can be used in the classifier

2.4. Inspecting CoAID_News dataset (5457 entries)

- True entries (4532) are very noisy and of not the greatest quality. Mostly they are articles title that don't have any factual claims in them and are just introductions to bigger articles where the topic is expanded
- Fake claims (925) on the other hand are structured correctly and seem to be of good quality

In [8]:
```
# I am dropping all true entries from here and taking all the fake claims.
CoAID_news_picked = CoAID_news[CoAID_news['claim_veracity'] == 0]
```

3. Saving the dataset

In [9]:
```
CoAID_Final = pd.concat([CoAID_claim, CoAID_news])
CoAID_Picked = pd.concat([CoAID_claim_picked, CoAID_news_picked], ignore_index=True) # 1000 entries
```

In [10]:
```
# Change the column name
CoAID_Final.rename(columns={'title': 'claim'}, inplace=True)
CoAID_Picked.rename(columns={'title': 'claim'}, inplace=True)
```

In [11]:
```
CoAID_Picked.to_csv('CoAID_Picked.csv', encoding='utf-8')
CoAID_Final.to_csv('CoAID_Final.csv', encoding='utf-8')
```

C.2.5 Kaggle.ipynb

KAGGLE Dataset

Headlines and bodies of news articles.

Necessery Imports

In [ ]:
```
import pandas as pd
import numpy as np
import plotly.express as px
import plotly
pd.set_option('display.max_rows', 500, "display.max_colwidth", None)
plotly.offline.init_notebook_mode(connected=True)
import warnings
warnings.filterwarnings('ignore')
```

1. Loading the datasets

In [ ]:
```
# Loading the dataset and adding claim_veracity column
KAGGLE_true = pd.read_csv('Initial_datasets/Kaggle_True.csv', low_memory=False)
KAGGLE_true['claim_veracity'] = 1
KAGGLE_false = pd.read_csv('Initial_datasets/Kaggle_Fake.csv', low_memory=False)
KAGGLE_false['claim_veracity'] = 0

KAGGLE_df = pd.concat([KAGGLE_true, KAGGLE_false], ignore_index=True)
print("size od dataset: ", KAGGLE_df.shape)
```

2. Formatting the dataset

2.1 Dropping unnessery columns

In [ ]:
```
KAGGLE_df = KAGGLE_df.drop(['text', 'date'], 1)
KAGGLE_df.head(5)
```

2.2 Removing duplicates

In [ ]:
```
# Removind duplicates (around 120)
print(KAGGLE_df.shape[0])
KAGGLE_df = KAGGLE_df.drop_duplicates(subset='title', keep='first')
print(KAGGLE_df.shape[0])
```

2.3 Inspecting the subject

In [ ]:
```
fig = px.histogram(KAGGLE_df, x='subject')
fig.update_layout(bargap=0.2)
fig.show()
```

2.4 Any special characters

In [ ]:
```
# Droping titles with @
KAGGLE_df = KAGGLE_df.drop(KAGGLE_df[KAGGLE_df['title'].str.contains('@', regex=True)].index)
```

In [ ]:
```
# Droping titles with \n
KAGGLE_df = KAGGLE_df.drop(KAGGLE_df[KAGGLE_df['title'].str.contains('\n', regex=True)].index)
```

In [ ]:
```
# Droping titles with links ('http')
KAGGLE_df = KAGGLE_df.drop(KAGGLE_df[KAGGLE_df['title'].str.contains('http', regex=True)].index)
```

In [ ]:
```
# Droping titles with # -> Mostly very weird entries that are not reliable (~400)
KAGGLE_df = KAGGLE_df.drop(KAGGLE_df[KAGGLE_df['title'].str.contains('#', regex=True)].index)
```

In [ ]:
```
# Titles containing ' & ' -> Replace with ' and ' instead (25)
KAGGLE_df['title'] = KAGGLE_df['title'].str.replace(' & ',' and ')

# Drop the rest of the titles containing &
KAGGLE_df = KAGGLE_df.drop(KAGGLE_df[KAGGLE_df['title'].str.contains('&', regex=True)].index)
```

In [ ]:
```
# Droping titles with ; -> Hard to trust such titles
KAGGLE_df = KAGGLE_df.drop(KAGGLE_df[KAGGLE_df['title'].str.contains(';', regex=True)].index)
```

In [ ]:
```
KAGGLE_df.shape[0]
```

2.5 Other observations

- Looking for frequent words that are suspicious and unnessery

In [ ]:
```
# Take most common words

from collections import Counter
print(Counter(" ".join(KAGGLE_df["title"]).split()).most_common(200))
```

In [ ]:
```
def replace_words(replace, replace_with):
    KAGGLE_df['title'] = np.where(KAGGLE_df['title'].str.lower().str.contains(replace, regex=False),
                                  KAGGLE_df['title'].str.lower().str.replace(replace, replace_with, regex=Fals
                                  KAGGLE_df['title'])
```

In [ ]:
```
# Deleting various inside brackets info that doesn't affect these listings
# [] brackets
KAGGLE_df['title'] = np.where(KAGGLE_df['title'].str.contains(r'\[[A-Za-z\/ ,.\'0-9]*\]', regex=True),
                              KAGGLE_df['title'].str.replace(r'\[[A-Za-z\/ ,.\'0-9]*\]', '', regex=True),
                              KAGGLE_df['title'])

# () brackets
KAGGLE_df['title'] = np.where(KAGGLE_df['title'].str.contains(r'\([A-Za-z\/ ,.\'0-9]*\)', regex=True),
                              KAGGLE_df['title'].str.replace(r'\([A-Za-z\/ ,.\'0-9]*\)', '', regex=True),
                              KAGGLE_df['title'])

# Deleting other words that don't bring much to the table
replace_words('factbox - ', '')
replace_words('factbox:', '')
replace_words('wow!', '')
replace_words('wow', '')
replace_words('exlusive:', '')
replace_words('exlusive - ', '')
replace_words('exlusive ', '')
replace_words('watch: ', '')
```

- Many entries are quating some twitter posts in an unusual way that doesn't ressemble a "normal claim"

In [ ]:
```
# Delete the "on Twitter" entries (212)
KAGGLE_df = KAGGLE_df.drop(KAGGLE_df[KAGGLE_df['title'].str.lower().str.contains('on twitter', regex=True)].index)
```

2.6. TRUE/FALSE Distribution after formatting

In [ ]:
```
fig = px.histogram(KAGGLE_df, x='claim_veracity').update_xaxes(categoryarray=[1, 0])
fig.update_layout(bargap=0.2)
fig.show()
```

2.7. Sentence Length

- The lower character length sentences seem to bad. Very non-informative - EXAMINE

In [ ]:
```
KAGGLE_df['title_count'] = KAGGLE_df['title'].str.len()

fig = px.histogram(KAGGLE_df, x='title_count') # With log scale to see a better distribution
fig.update_layout(bargap=0.2)
fig.show()
```

In [ ]:
```
KAGGLE_df_1 = KAGGLE_df[KAGGLE_df['title_count'] <= 35]  # 0 - 30 -> DELETE ALL THESE (64)
KAGGLE_df_2 = KAGGLE_df[(KAGGLE_df['title_count'] > 35) & (KAGGLE_df['title_count'] <= 50)]  # 35 - 50 (1377)
KAGGLE_df_3 = KAGGLE_df[(KAGGLE_df['title_count'] > 50) & (KAGGLE_df['title_count'] <= 75)] # 50 - 75 (20886)
KAGGLE_df_4 = KAGGLE_df[(KAGGLE_df['title_count'] > 75) & (KAGGLE_df['title_count'] <= 100)] # 75 - 100 (11502)
KAGGLE_df_5 = KAGGLE_df[KAGGLE_df['title_count'] > 100] # 100 - 180 (4106)
KAGGLE_df_6 = KAGGLE_df[KAGGLE_df['title_count'] > 180] # 180 - ___ (113) Don't like the quality of these entries
```

3. Making selected dataset

Making a selected Kaggle dataset with entries from each category based on sentence length and their claim veracity

- 8 different ranges of characters length that split the dataset into 8 groups

In [ ]:
```
# 800 random entries from each category will be chosen (400 from true and 400 from fake)

KAGGLE_df_true = KAGGLE_df[KAGGLE_df['claim_veracity'] == 1]
KAGGLE_df_false = KAGGLE_df[KAGGLE_df['claim_veracity'] == 0]
# df_list = []

dfs = []
for df in [KAGGLE_df_true, KAGGLE_df_false]:
    df['char_category'] = pd.qcut(df['title_count'], 8, labels=[0, 1, 2, 3, 4, 5, 6, 7])
    for category in [0, 1, 2, 3, 4, 5, 6, 7]:
        category_df = df[df['char_category'] == category]
        dfs.append(category_df.sample(400, random_state=1))

KAGGLE_picked = pd.concat(dfs)
```

4. Saving Datasets

In [ ]:
```
# Rename title to claim
KAGGLE_df.rename(columns={'title': 'claim'}, inplace=True)
KAGGLE_picked.rename(columns={'title': 'claim'}, inplace=True)
```

4.1. Whole KAGGLE

In [ ]:
```
KAGGLE_df = KAGGLE_df.sample(frac=1).reset_index(drop=True).drop(['subject', 'title_count'], axis=1)
KAGGLE_df.to_csv('KAGGLE_Final.csv', encoding='utf-8')
```

4.2. Picked KAGGLE

In [ ]:
```
KAGGLE_picked = KAGGLE_picked.sample(frac=1).reset_index(drop=True).drop(['subject', 'title_count', 'char_category
KAGGLE_picked.to_csv('KAGGLE_Picked.csv', encoding='utf-8')
```

C.2.6 TwitterFakeNews.ipynb

TwitterFakeNews

Large scale dataset of 200k+ tweets that were labelled automatically to be true/false. Tweets that are issued by accounts known for spreading false news are considered false (all of them) and vice-versa for trustworthy accounts

https://www.mdpi.com/1999-5903/13/5/114

Neccessery imports

In [1]:
```
import pandas as pd
import numpy as np
import plotly.express as px

pd.set_option("display.max_rows", None, "display.max_columns", None)
pd.options.display.max_colwidth = 200
```

1. Loading the dataset

tweet_fake - 1 (Fake), 0 (True)

In [2]:
```
picked_columns = ['tweet__text', 'tweet__contains_hashtags', 'tweet__nr_of_hashtags', 'tweet__sent_tokenized_text'
Auto_df = pd.read_csv('Initial_datasets/data.csv', sep=';', usecols = picked_columns , low_memory=False)

Auto_df.head(3)
```

Out[2]:

	tweet__retweet_count	tweet__favorite_count	tweet__fake	tweet__sent_tokenized_text	tweet__text	tweet__nr_of_sentences
0	32	71	0	["Joe Buck ruined Brooks Koepka's U.S. Open kiss with a super-awkward mistake "]	Joe Buck ruined Brooks Koepka's U.S. Open kiss with a super-awkward mistake https://t.co/UO1GclUPU6	1
1	67	55	0	['Two engaged doctors found bound and slain in luxury Boston penthouse.', 'Man in custody after shootout. ']	Two engaged doctors found bound and slain in luxury Boston penthouse. Man in custody after shootout. https://t.co/A22ivfzonQ	2
2	0	1	1	['DROP EVERYTHING and watch these four urgent, must-see #Documentries ']	DROP EVERYTHING and watch these four urgent, must-see #Documentries https://t.co/TiFhU40nWq https://t.co/1dk8PZWFYC	1

2. Claim Insepctions & Formatting

2.1. URLs

In [3]:
```
# The one that start with a link are dropped (41 entries)
Auto_df = Auto_df.drop(Auto_df.loc[Auto_df['tweet__text'].str.lower().str.startswith('http')].index)

#The ones that have link in the middle are also dropped (8500) (too risky - most of them are faulty/weird)
Auto_df = Auto_df.drop(Auto_df.loc[Auto_df['tweet__text'].str.lower().str.split().str[-1].str.contains('http') ==

#The ones that have a link at the end (180.000+) are having their link\links deleted
f = lambda x: ' '.join([item for item in x.split() if 'http' not in item])
Auto_df["tweet__text"] = Auto_df["tweet__text"].apply(f)

print(Auto_df.shape[0])
```

```
188773
```

2.2. Hashtags - Dropping hashtags

Because the dataset is big we can drop the hashtags and still end up with a very big dataset. The hashtag entries are not reliable, hence the decision to not examine them any further

In [4]:
```
print("size of entries with hashtags: ", Auto_df[Auto_df['tweet__contains_hashtags'] == True].shape[0])
hash_df = Auto_df[Auto_df['tweet__contains_hashtags'] == True]

Auto_df = Auto_df.drop(Auto_df[Auto_df['tweet__contains_hashtags'] == True].index)
```

```
size of entries with hashtags:  25098
```

2.3. Duplicates

In [5]:
```
# Dropping duplicate entries (over 25.0000 rows)
Auto_df = Auto_df.drop_duplicates(subset='tweet__text', keep='first')
```

2.4. @ts

In [6]:
```
# Droping entries with mentions (@) as there is no real way to transorm them to normal text (around 15.000)
Auto_df = Auto_df.drop(Auto_df.loc[Auto_df['tweet__text'].str.lower().str.contains('@')].index)
```

2.5. Other Special Characters

- &amp
- ;
- non ascii characters (like emojis for example)

In [7]:
```
# drop entries containing &amp (1500)
Auto_df = Auto_df.drop(Auto_df[Auto_df['tweet__text'].str.contains('&amp')].index)
Auto_df = Auto_df.replace(';',',', regex=True)
Auto_df['tweet__text'] = Auto_df['tweet__text'].astype(str).apply(lambda x: x.encode('ascii', 'ignore').decode('as
```

2.6. Nr of Sentences

- Deleting everything that is over 2 sentences long (around 1500 entries)

In [8]:
```
Auto_df = Auto_df.drop(Auto_df[Auto_df['tweet__nr_of_sentences'] > 2].index)
```

2.7. Tweet Favorite/Retweet Count - Maybe take only the ones that are above some tweet fav count

- Droping the tweets that have 0 fav and retweet count (Not really worth checking them)

In [9]:
```
Auto_df = Auto_df.drop(Auto_df[Auto_df['tweet__favorite_count'] == 0].index)
Auto_df = Auto_df.drop(Auto_df[Auto_df['tweet__retweet_count'] == 0].index)
```

2.8. VIDEO entries

- Many entries have '- video' at the end of their claims. Drop them

In [10]:
```
Auto_df = Auto_df.drop(Auto_df[Auto_df['tweet__text'].str.contains('- video')].index)
```

3. True/Fake Distribution

In [ ]:
```
fig = px.histogram(Auto_df, x='tweet__fake')
fig.update_layout(bargap=0.2)
fig.show()
```

4. Saving new data

In [12]:
```
# Droping unncessery columns
Auto_df.drop(['tweet__retweet_count', 'tweet__favorite_count', 'tweet__sent_tokenized_text', 'tweet__nr_of_sentenc
```

- Spliting formatted data into fake and true entries that will be fed to the google verification algorithm, where each entry will get a levensthein distance score based on inputing the claim in google search

In [13]:
```
true_df = Auto_df[Auto_df['tweet__fake'] == 0]
fake_df = Auto_df[Auto_df['tweet__fake'] == 1]

true_df.to_csv('Auto_Format_True.csv', sep=';', encoding='utf-8')
fake_df.to_csv('Auto_Format_Fake.csv', sep=';', encoding='utf-8')
```

In [14]:
```
Auto_df.to_csv('TwitterFakeNews_Final.csv', encoding='utf-8')
```

5. Examining Google Verified Data

- These datasets went through a "google verification" method where each claim was inputted to the google search engine, and the levensthein score between the claim and the results were calculated. The idea is that the entries with higher average levensthein score are not as trustworthy as the ones with lower distance. This suggest that if the query has many similar results in google it should be considered true, and false otherwise. This was done to this noisy dataset as a layer of verification to their automatic labelling

5.1. Loading the datasets

In [15]:
```
Auto_true_score = pd.read_csv('Formatted_datasets/Auto_Format_True_Score.csv', index_col=0, low_memory=False)
Auto_false_score = pd.read_csv('Formatted_datasets/Auto_Format_False_Score.csv', index_col=0, low_memory=False)

# Delete the entries with leven_score = 0
Auto_true_score = Auto_true_score.drop(Auto_true_score.loc[Auto_true_score['leven__score'] <= 0].index)
Auto_false_score = Auto_false_score.drop(Auto_false_score.loc[Auto_false_score['leven__score'] <= 0].index)
```

5.2. Sentence Length correlation to score

- For True entries:

In [16]:
```
Auto_true_score['tweet__count'] = Auto_true_score['tweet__text'].str.len()
Auto_true_score['tweet__count'].corr(Auto_true_score['leven__score'])
```

Out[16]:
```
0.9132956515728649
```

- For False entries:

In [17]:
```
Auto_false_score['tweet__count'] = Auto_false_score['tweet__text'].str.len()
Auto_false_score['tweet__count'].corr(Auto_false_score['leven__score'])
```

Out[17]:
```
0.9053585456499308
```

As we can see there is a big correaltion between the tweet count and the levesthein score calculated from the google verifier. To adjust for that a new column is created

5.3. Adding Leven_Score/Tweet_Count column

- Adjusted Levensthein score to adjust for the length of the tweet. The leven_score alone was biased

In [18]:
```
Auto_true_score['adjusted_leven'] = Auto_true_score['leven__score'] / Auto_true_score['tweet__count']
Auto_false_score['adjusted_leven'] = Auto_false_score['leven__score'] / Auto_false_score['tweet__count']
```

In [19]:
```
print(Auto_true_score['tweet__count'].corr(Auto_true_score['adjusted_leven']))
print(Auto_false_score['tweet__count'].corr(Auto_false_score['adjusted_leven']))
```

```
-0.08080815730036466
-0.385536888505841
```

As we can see strong correlation is now gone, making the results more trustworthy. The lower adjusted_leven, the more chance the tweet__text label is actually true

5.4 Picking Best Entries Based On Sentence Length

- For True entries:

Pick 3500 entries with the best (lowest) adjusted_leven score -> This indicates that is simillarity between the claim and the google search query results

In [ ]:
```
# Take 3000 best rated entries from all
Final_Auto_True = Auto_true_score.sort_values('adjusted_leven').head(3500)
Final_Auto_True

fig = px.histogram(Final_Auto_True, x='tweet__count', title="Best 3500 claims and their characters length")
fig.update_layout(bargap=0.2)
fig.show()

fig = px.histogram(Auto_true_score, x='tweet__count', title="Characters length distribution of all true entries")
fig.update_layout(bargap=0.2)
fig.show()
```

Conclusion

We can see claims mostly at around 60 characters long, and not much claims that are over 80 characters long. However from the distribution of the whole dataset we can see that big portion of the claims are longer entries with around 100+ characters. Whereas on manual inspection shorter claims appear to be make more sense and be more reliable I decided to include

In [21]:
```
# Remaining dataset: (droping rows that are already considered valid)
Reamaining_df = Auto_true_score.drop(Final_Auto_True.index)

print("4 different ranges of characters length that split the dataset into 4 groups: \n\n",
      pd.cut(Reamaining_df['tweet__count'], 4).unique())

Reamaining_df['char__category'] = pd.cut(Reamaining_df['tweet__count'], 4, labels=[0, 1, 2, 3])
```

```
4 different ranges of characters length that split the dataset into 4 groups:

 [(89.75, 118.0], (61.5, 89.75], (33.25, 61.5], (4.887, 33.25]]
Categories (4, interval[float64]): [(4.887, 33.25] < (33.25, 61.5] < (61.5, 89.75] < (89.75, 118.0]]
```

Category Inspection conclusions:

- **0 (4 - 33 characters)** -> Entries don't tell anything and **should not** be classified as true or false. They are mostly empty headlines. Do not include
- **1 (34 - 61 characters)** -> Entries seem very reliable and the ones with the best score show most potential. Include best 500
- **2 (62 - 89 characters)** -> These also show promise and look reliable upon initial inspection. Include best 500
- **3 (90 - 118 chatacters)** -> Same with these. Include best 500

In [22]:
```
# Adding additional 1500 enries from categories 1-3 (500 best in each category)
for category in [1,2,3]:
    top500_df = Reamaining_df[Reamaining_df['char__category'] == category].sort_values('adjusted_leven').head(500)
    Final_Auto_True = pd.concat([Final_Auto_True, top500_df])
```

- For Fake entries:

For fake entries we will pick claims that achieved the biggest levensthein distance, meaning entries that had the biggest discrepancy between the claim and the google search results

In [ ]:
```
fig = px.histogram(Auto_false_score, x='tweet__count', title="Characters length distribution of all fake entries")
fig.update_layout(bargap=0.2)
fig.show()
```

Upon inspection we can't just simply take the worst values from leven__score or adjusted_leven as the values are biased towards claims with little characters or many characters accordingly. Therefore as with true instances, the entries are grouped by character length beforehand .

In [24]:
```
print("4 different ranges of characters length that split the dataset into 4 groups: \n\n",
      pd.cut(Auto_false_score['tweet__count'], 4).unique())

Auto_false_score['char__category'] = pd.cut(Auto_false_score['tweet__count'], 4, labels=[0, 1, 2, 3])
```

```
4 different ranges of characters length that split the dataset into 4 groups:

 [(88.0, 116.0], (32.0, 60.0], (60.0, 88.0], (3.888, 32.0]]
Categories (4, interval[float64]): [(3.888, 32.0] < (32.0, 60.0] < (60.0, 88.0] < (88.0, 116.0]]
```

Category Inspection conclusions:

- **0 (4 - 32 characters)** -> Entries don't tell anything and **should not** be classified as true or false. They are mostly empty headlines. Do not include
- **1 (33 - 60 characters)** -> Entries seem very reliable and the ones with the best score show most potential. Include best 25%
- **2 (61 - 88 characters)** -> These also show promise and look reliable upon initial inspection. Include best 25%
- **3 (89 - 116 chatacters)** -> Same with these. Include best 25%

In [25]:
```
dfs = []
for category in [1,2,3]:
    category_df = Auto_false_score[Auto_false_score['char__category'] == category].sort_values('adjusted_leven', a
    dfs.append(category_df.head(int(len(category_df) * (0.20))))

Final_Auto_False = pd.concat(dfs)
```

6. Saving Final Dataset

In [15]:
```
# Switch the tweet_fake labels to reflect not wether the tweet is fake but wether if it's true
Final_Auto_False = Final_Auto_False.assign(tweet__fake=0)
Final_Auto_True = Final_Auto_True.assign(tweet__fake=1)
```

In [16]:
```
# Merging True and False datasets together
Final_df = pd.concat([Final_Auto_False, Final_Auto_True])
```

In [17]:
```
# Dropping unncessery columns, renamining and rearranging remaining one
Final_df = Final_df.sample(frac=1).reset_index(drop=True).drop(['leven__score', 'tweet__count', 'adjusted_leven',
Final_df.rename(columns={'tweet__fake': 'claim_veracity', 'tweet__text': 'claim'}, inplace=True)
Final_df = Final_df[['claim', 'claim_veracity']]
```

In [18]:
```
Final_df.to_csv('TwitterFakeNews_Picked.csv', encoding='utf-8')
```

C.2.7 Final_Dataset.ipynb

Data preperation for classification model

This notebook creates a final dataset(s) that will be used when training/testing model for binary classifiaction of fake/true claims.
According training/testing/validation datasets will be created later on

Necessery imports

In [1]:
```
import pandas as pd
import numpy as np
import plotly.express as px
import plotly.offline as pyo
import plotly.graph_objs as go
pyo.init_notebook_mode()

pd.set_option("display.max_rows", None, "display.max_columns", None)
pd.set_option('display.max_colwidth', None)
pd.options.display.max_colwidth = 200
```

1. Loading Datasets

In [2]:
```
TwitterFakeNews_df = pd.read_csv('../TwitterFakeNews/Formatted_datasets/TwitterFakeNews_Picked.csv', index_col=0,
FEVER_df = pd.read_csv('../FEVER/Formatted_datasets/FEVER_Picked.csv', index_col=0, low_memory=False)
LIAR_df = pd.read_csv('../LIAR/Formatted_datasets/LIAR_Final.csv', index_col=0, low_memory=False)
CoAID_df = pd.read_csv('../CoAID/Formatted_datasets/CoAID_Picked.csv', index_col=0, low_memory=False)
CovidFakeNews_df = pd.read_csv('../CovidFakenews/Formatted_datasets/CovidFakeNews_Final.csv', index_col=0, low_mem
KAGGLE_df = pd.read_csv('../KAGGLE/Formatted_datasets/KAGGLE_Picked.csv', index_col=0, low_memory=False)
```

In [3]:
```
# Putting labels on datasets for analysis purposes
TwitterFakeNews_df['dataset'] = 'TwitterFakeNews'
FEVER_df['dataset'] = 'FEVER'
LIAR_df['dataset'] = 'LIAR'
CoAID_df['dataset'] = 'CoAID'
CovidFakeNews_df['dataset'] = 'CovidFakeNews'
KAGGLE_df['dataset'] = 'Kaggle'
```

2. Merging them together

In [4]:
```
df = pd.concat([TwitterFakeNews_df, FEVER_df, LIAR_df, CoAID_df, CovidFakeNews_df, KAGGLE_df])
df = df.reset_index(drop=True)
```

3. Data Formatting

Last data formatting to check for duplicates/faulty entries

3.1. Removing duplicates

In [5]:
```
# Dropping duplicate entries (over 300 entries)
df = df.drop_duplicates(subset='claim', keep='first')
```

3.2 Non-Ascii characters removed

In [6]:
```
# Deleting non-ascii characters from the strings
df['claim'] = df['claim'].astype(str).apply(lambda x: x.encode('ascii', 'ignore').decode('ascii')) # remove charac
```

4. Dataset analysis

4.1. Characters Length distributions

In [ ]:
```
df['claim_count'] = df['claim'].str.len()

fig = px.histogram(df, x='claim_count', title="Characters length distribution of all true entries initially")
fig.update_layout(bargap=0.2)
fig.show()
```

We can see that there are still some entries with very long claims (over 500 characters). In order for the BERT model to work I need to trim the dataset to only take into account smaller claims.

In [8]:
```
# Removing entries with claims over 250 characters (690 of them), as I want to focus on shorter statements.
df.drop(df.loc[df['claim_count'] > 250].index, inplace=True)
```

In [ ]:
```
fig = px.histogram(df, x='claim_count', title="Characters length distribution of all true entries after deleting l
fig.update_layout(bargap=0.2)
fig.show()
```

As we can see the distribution is more normalised now

4.2. Dataset distributions

In [ ]:
```
fig = px.pie(df, names = 'dataset', color = 'dataset', color_discrete_sequence=px.colors.qualitative.Pastel)
fig.update_traces(text = df['dataset'].value_counts(), textinfo = 'label+percent')
fig.update_traces(hole=.4, hoverinfo="label+percent+name")
fig.update_layout(
    annotations=[dict(text='Total Size: 38,659', x=0.50, y=0.5, font_size=15, showarrow=False)])
fig.show()
```

4.3. True/Fake Distribution

In [ ]:
```
# fig = px.histogram(df, x='claim_veracity')
# fig.update_layout(bargap=0.2)
# fig.show()


fig = px.pie(df["claim_veracity"].astype(bool), names = 'claim_veracity', color = 'claim_veracity', color_discrete
fig.update_traces(text = df['dataset'].value_counts(), textinfo = 'label+percent', textfont_size=15)
fig.show()

# 19982 false and 18677 true entries
```

5. Saving the dataset

Splitting each dataset so that 20% of it will be used in the testing part

In [ ]:
```
from sklearn.model_selection import train_test_split

TwitterFakeNews_df = df[df['dataset'] == 'twitter fake news']
FEVER_df = df[df['dataset'] == 'FEVER']
LIAR_df = df[df['dataset'] == 'LIAR']
CoAID_df = df[df['dataset'] == 'CoAID']
CovidFakeNews_df = df[df['dataset'] == 'Covid fake news']
KAGGLE_df = df[df['dataset'] == 'Kaggle']

training_dfs = []
testing_dfs = []
for tmp_df in [TwitterFakeNews_df, FEVER_df, LIAR_df, CoAID_df, CovidFakeNews_df, KAGGLE_df]:
    train, test = train_test_split(tmp_df, test_size=0.2)
    training_dfs.append(train)
    testing_dfs.append(test)

testing_df = pd.concat(testing_dfs)
training_df = pd.concat(training_dfs)
```

In [ ]:
```
def saveDataset(df, csv_name):
    df = df.sample(frac=1)
    df = df.reset_index(drop=True)
    df = df[['claim', 'claim_veracity']] # dropping unnessery columns
    df.to_csv(csv_name, encoding='utf-8')

saveDataset(training_df, 'training.csv')
saveDataset(testing_df, 'testing.csv')
```

In [ ]:
```
whole_dataset = pd.concat([training_df, testing_df])
saveDataset(whole_dataset, 'dataset.csv')
```

C.3 Models Implementation and Evaluation

C.3.1 Models_Implementation.ipynb

Creating, fine tunining and experimenting with BERT-based models for fake news classifiaction

This is the main Jupyter Notebook for creating, fine-tuning, evaluating and experimenting with different BERT-based models implemented in the project: ''A thorough attempt to enhance fake news detection through unbiased dataset, explainability and BERT-based models'' by Jan Marczak.

The notebook is intended to be run on Google Colab: https://colab.research.google.com to make use of their available GPUs. The notebook is intended to run in order, hence all proceeding cells should be executed before running a particular block of code. The whole file is organised in sections, so that each part of this notebook has a clear usage:

- Chapter 1 is responsible for setting up the GPU and installing necessery libraries for later implementation
- Secttions 2 and 3 are related to data preprocessing
- Chapters from 4 to 6 define model architectures classes, initialise other model components and create both training and testing loops.
- Sections from 7 to 10 provide the main code for using previously defined functions and define best-suited hyperparameters, evaluate different BERT-based models and neural networks.
- Section 11 adds LIME explainability on top of some trained model

The following code and process was inspired by tutorials listed below. However most of their ideas were largely refactored for particular use cases of this project

- https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FIdYFlpcX#scrollTo=GI0iOY8zvZzL
- https://github.com/prathameshmahankal/Fake-News-Detection-Using-BERT/blob/main/notebooks/train.ipynb
- https://chriskhanhtran.github.io/posts/bert-for-sentiment-analysis/#1-install-the-hugging-face-library

1. Setup

In order to run this code it is highly recommended to:

- Follow a README provided with submission
- Run this code in Google Colab
- Upload 'training.csv', 'testing.csv' and 'dataset.csv' in google colab through side bar: `Files -> Upload to Session Storage -> Pick 3 mentioned datasets`
- Use Colab GPU as said in 1.1.
- If the user is inactive, Google Colab can terminate the session and all these steps would have to be repeated to run this code

1.1. Using availble GPUs offered by Google Colab

In order to run this code without problems, a GPU should be added by first selecting:

`Edit -> Notebook Settings -> Hardware accelerator -> (GPU)`

from the top menu, and then running the code below:

In [ ]:
```
import torch

# Later the data will be loaded to this device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

print('Using ', device) # the code should say 'Using cuda'
```

1.2. Installing and importing necessery libraries

In [ ]:
```
!pip install transformers
!pip install lime
!pip install sentencepiece
```

In [ ]:
```
import numpy as np
import pandas as pd
import time
import datetime
import random
import csv

from matplotlib import pyplot as plt
from transformers import BertTokenizer, RobertaTokenizer, AutoTokenizer, BertModel, RobertaModel, AutoModel, Albe
from torch.utils.data import TensorDataset, random_split, DataLoader, RandomSampler, SequentialSampler, Sampler
import torch
import torch.nn as nn
from torch.optim import Optimizer, AdamW
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay, roc_curve, f1_score, recall_score, precision_score, accuracy_

import lime
from lime.lime_text import LimeTextExplainer

from google.colab import drive
```

2. Reading Datasets

Make sure to upload datasets before running the code!

- There are multiple ways to read CSV files in Google Colab, and since the file is deleted after refreshing runtime files, it has to be reloaded upon each execution of the notebook.
- To load the dataset look at the left side menu and select:

`Files -> Upload to session storage -> find & choose 'dataset.csv', 'training.csv' and 'testing.csv'`

In [ ]:
```
def read_data(path: str):
  """
  Read the data from a file into a dataframe
  """
  print("Reading the dataset...")
  try:
    df = pd.read_csv(path, index_col = 0)
    print("\nThe data of size ", df.shape, ' has been loaded.')
    return df
  except:
    print('ERROR: Could not read the file at the path: ', path, '\n\nMake sure you load the dataset first and nam
    return pd. DataFrame() #return empty dataframe
```

In [ ]:
```
def get_data_as_arrays(df: pd.DataFrame):
  """
  Split the data into arrays of claims and labels
  """
  claims = df.claim.values
  labels = df.claim_veracity.values
  return claims, labels
```

3. Dataset Preprocessing

The dataset will now be transformed into the format that BERT-like models can be trained on.

BERT has some formatting requirements that need to be satisfied before moving forward:

- Add special [CLS] and [SEP] tokens to the start and end of each sentence.
- Pad & truncate all sentences to a single **fixed** length. Padding is done with a special [PAD] token.
- Explicitly differentiate real tokens form padding tokens with the "attention mask".

3.1. Model Tokenizer

A tokenizer is in charge of preparing the inputs for a model. Tokenizer splits the sentence into tokens, which are then mapped to their index in the tokenizer vocabulary. Each pre-trained BERT-like model has its own tokenizer.

In [ ]:
```
def load_model_tokenizer(model_name: str, model_type: str):
  """
  Load correct model tokenizer, based on the model_type
  """
  if model_type == 'bert':
      return BertTokenizer.from_pretrained(model_name)
  elif model_type == 'roberta':
      return RobertaTokenizer.from_pretrained(model_name)
  elif model_type == 'albert':
      return AlbertTokenizer.from_pretrained(model_name)
  elif model_type == 'bertweet':
      return AutoTokenizer.from_pretrained(model_name)
```

3.2 Ploting the distribution of tokenized data

For deciding the maximum length of the tokenized input BERT model will accept, the dataset can be first tokenized and their lengths presented on a graph. This will be used later, during defining hyperparameters

In [ ]:
```
def plot_tokenized_data(data, tokenizer):
    """
    First tokenize the whole dataset using a tokenizer,
    and then plot their tokenized input distribution.
    """
    if not data.empty:
        claims, labels = get_data_as_arrays(data) # Moving claims and labels to numpy arrays for easier preproces

        tokenized_inputs = []
        for claim in claims:
            # Tokenize the text and add `[CLS]` and `[SEP]` tokens.
            tokenized_input = tokenizer.encode(claim, add_special_tokens=True)
            tokenized_inputs.append(len(tokenized_input))

        # fixed bin size
        bins = np.arange(-100, 100, 5) # fixed bin size
        plt.xlim([min(tokenized_inputs)-5, max(tokenized_inputs)+5])

        plt.hist(tokenized_inputs, bins=bins, alpha=0.5)
        plt.title('Tokenized sentences lengths distribution')
        plt.xlabel('tokenized_input length (bin size = 5)')
        plt.ylabel('count')

        plt.show()
        print('min: ', min(tokenized_inputs), 'max: ', max(tokenized_inputs))
    else:
        print("training data not found - make sure to import it and name it training.csv")
```

3.3. Tokenization

Tokenization will handle most of the parsing and data preperation needed for feeding into BERT. The function below combines multiple steps:

1. Split the sentence into tokens.
2. Add the special `[CLS]` and `[SEP]` tokens.
3. Map the tokens to their IDs.
4. Pad or truncate all sentences to the same length.
5. Create the attention masks which explicitly differentiate real tokens from `[PAD]` tokens.

In [ ]:
```
def data_tokenization(claims: np.array, tokenizer: PreTrainedTokenizer, max_claim_length = 80):
  """
  Tokenize claims using a pre-trained tokenizer from Transofmer-like models
  """
  data_tokens = tokenizer.__call__(
    claims.tolist(),                  # Sentence to encode.
    add_special_tokens = True,        # Add '[CLS]' and '[SEP]'.
    padding = True,                   # Pad sentences.
    truncation = True,                # Truncate sentences.
    max_length = max_claim_length,    # Specify max length
    return_attention_mask = True,     # Construct attn. masks.
  )

  # Convert the lists into tensors.
  input_ids = torch.tensor(data_tokens['input_ids'])
  attention_masks = torch.tensor(data_tokens['attention_mask']) # differentiates padding from non-padding
  # labels = torch.tensor(labels)

  # return input_ids, attention_masks, labels
  return input_ids, attention_masks
```

3.4. Splitting the dataset into Training & Validation

Dividing up the training dataset into 85% for training and 15% for validation

In [ ]:
```
def create_training_validation_data(input_ids, attention_masks, labels, train_percentage = 0.85):
    """
    Split the training dataset into a training and validation dataset
    based on train_percentage
    """
    dataset = TensorDataset(input_ids, attention_masks, labels)

    train_size = int(train_percentage * len(dataset))
    validation_size = len(dataset) - train_size

    train_dataset, validation_dataset = random_split(dataset, [train_size, validation_size])

    print('{:>5,} training samples'.format(train_size))
    print('{:>5,} validation samples'.format(validation_size))

    return train_dataset, validation_dataset
```

3.5. Create DataLoaders

Dataloaders wrap an iterable around TensorDataset objects to enable easy access to the samples. This helps save on memory during the traning process because, unlike a for loop, with an iterator the entire dataset does not need to be loaded into memory

In [ ]:
```
def create_dataloader(dataset, isRandomSampler, batch_size = 32):
    """
    Create DataLoaders objects
    """
    if isRandomSampler:
        return DataLoader(
            dataset,
            sampler = RandomSampler(dataset),
            batch_size = batch_size
        )

    else:
        return DataLoader(
            dataset,
            sampler = SequentialSampler(dataset),
            batch_size = batch_size
        )
```

4. Creating BERT-based models and its components for fake news classifiaction

Because many different BERT-based models are created in this project, this section defines their model architectuers and necessary components.

4.1. Defining Models Architecure

This section defines model architectures for BERT, RoBERTa, ALBERT and BERTweet. They all follow the same structure, hence only section 4.1.1. about BERT architectures is explained with additional comments.

4.1.1. BERT

In [ ]:
```python
class BertForFakeNewsDetection(nn.Module):
  """
  BERT Model for classyfing fake news to true/false.
  """
  def __init__(self, freeze_model = False, model_name = 'bert-base-cased'):
    """
    @param    classifier: a torch.nn.Module classifier
    @param    freeze_model (bool): Set 'False' to fine-tune the BERT model
    @param    model_name: a name of pre-trained bert model
    """
    super(BertForFakeNewsDetection, self).__init__()
    # Instantiate BERT model
    self.bert = BertModel.from_pretrained(model_name) # Embedding layer
    input_size = self.bert.config.to_dict()['hidden_size']

    # Our classifier (feed forward neural network)
    self.classifier = nn.Sequential(
        nn.Dropout(0.2),             # Dropout regulatization method
        nn.Linear(input_size, 50),    # Hidden Layer
        nn.ReLU(),                    # Activation Function
        nn.Linear(50, 2),             # Output Layer
        nn.LogSoftmax(dim=1)          # Activation Function in the output
    )

    # Freeze the BERT model - True if you don't want to fine tune BERT
    if freeze_model:
        for param in self.bert.parameters():
            param.requires_grad = False


  def forward(self, input_ids, attention_mask):
    """
    Feed input to BERT and the classifier to compute logits.
    @param    input_ids (torch.Tensor): an input tensor with shape (batch_size, max_length)
    @param    attention_mask (torch.Tensor): a tensor that hold attention mask
                                             information with shape (batch_size, max_length)
    @return   logits (torch.Tensor): an output tensor with shape (batch_size, num_labels)
    """

    # Feed input to bert and extract last hidden state of the [CLS] token, that is used for classificaiton task.
    _, last_cls_token = self.bert(input_ids = input_ids, attention_mask = attention_mask, return_dict = False)

    # Feed input to classifier to compute logits ()
    logits = self.classifier(last_cls_token)

    return logits
```

4.1.2. RoBERTa

In [ ]:
```python
class RobertaForFakeNewsDetection(nn.Module):

  def __init__(self, freeze_model = False, model_name = 'roberta-base'):
    super(RobertaForFakeNewsDetection, self).__init__()
    self.roberta = RobertaModel.from_pretrained(model_name)
    input_size = self.roberta.config.to_dict()['hidden_size']

    self.classifier = nn.Sequential(
        nn.Dropout(0.2),
        nn.Linear(input_size, 50),
        nn.ReLU(),
        nn.Linear(50, 2),
        nn.LogSoftmax(dim=1)
    )

    if freeze_model:
        for param in self.roberta.parameters():
            param.requires_grad = False

  def forward(self, input_ids, attention_mask):
    _, last_cls_token = self.roberta(input_ids = input_ids, attention_mask = attention_mask, return_dict = False)
    logits = self.classifier(last_cls_token)
    return logits
```

4.1.3. ALBERT

In [ ]:
```
class AlbertForFakeNewsDetection(nn.Module):

  def __init__(self, freeze_model = False, model_name = 'albert-base-v2'):
    super(AlbertForFakeNewsDetection, self).__init__()
    self.albert = AlbertModel.from_pretrained(model_name)
    input_size = self.albert.config.to_dict()['hidden_size']

    self.classifier = nn.Sequential(
        nn.Dropout(0.2),
        nn.Linear(input_size, 50),
        nn.ReLU(),
        nn.Linear(50, 2),
        nn.LogSoftmax(dim=1)
    )

    if freeze_model:
        for param in self.albert.parameters():
            param.requires_grad = False

  def forward(self, input_ids, attention_mask):
    _, last_cls_token = self.albert(input_ids = input_ids, attention_mask = attention_mask, return_dict = False)
    logits = self.classifier(last_cls_token)
    return logits
```

4.1.4. BERTweet

In [ ]:
```
class BertweetForFakeNewsDetection(nn.Module):

  def __init__(self, freeze_model = False, model_name = 'bertweet-base'):
    super(BertweetForFakeNewsDetection, self).__init__()
    self.bertweet = AutoModel.from_pretrained(model_name)
    input_size = self.bertweet.config.to_dict()['hidden_size']

    self.classifier = nn.Sequential(
        nn.Dropout(0.2),
        nn.Linear(input_size, 50),
        nn.ReLU(),
        nn.Linear(50, 2),
        nn.LogSoftmax(dim=1)
    )

    if freeze_model:
        for param in self.bertweet.parameters():
            param.requires_grad = False

  def forward(self, input_ids, attention_mask):
    _, last_cls_token = self.bertweet(input_ids = input_ids, attention_mask = attention_mask, return_dict = Fals
    logits = self.classifier(last_cls_token)
    return logits
```

4.1.5. Create specific model

In [ ]:
```
def create_model(model_type, model_name, freeze_model = False):
    """
    Create a specific model instance based on the model_type and model_name
    """
    if model_type == 'bert':
        return BertForFakeNewsDetection(freeze_model, model_name)
    elif model_type == 'roberta':
        return RobertaForFakeNewsDetection(freeze_model, model_name)
    elif model_type == 'albert':
      return AlbertForFakeNewsDetection(freeze_model, model_name)
    elif model_type == 'bertweet':
        return BertweetForFakeNewsDetection(freeze_model, model_name)
```

4.2. Optimizer

Optimizers are algorithms or methods used to minimize the loss function or to maximise the efficieny od produciton. Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

Adam optimizer is one of the most popular and famous gradient descent optimization algorithms. It is a method that computes adaptive learning rates for each parameter. It stores both the decaying average of the past gradients , similar to momentum and also the decaying average of the past squared gradients. Loshchilov and Hutter proposed a variation, AdamW, which decouples weight decay from gradient computationx

For the purposes of fine-tuning, the authors of BERT recommend choosing from the following values (from Appendix A.3 of the BERT paper):

- **Batch size:** 16, 32
- **Learning rate (Adam):** 5e-5, 3e-5, 2e-5
- **Number of epochs:** 2, 3, 4

I will choose:

- Batch size: 32 (This is already a batch size of dataloaders)
- Learning rate: 2e-5
- Epochs: 4 (later this will be adjusted)

In [ ]:
```
def create_AdamW_optimizer(model: nn.Module, lr = 2e-5, eps = 1e-8):
  """
  Create an instance of an AdamW optimizer with specified parameters
  """
  return AdamW(model.parameters(), lr = lr, eps = eps)
```

4.3. Learning Rate Scheduler

- A Learning rate schedule is a predefined framework that adjusts the learning rate between epochs or iterations as the training progresses.
- Early in the training, the learning rate is set to be large in order to reach a set of weights that are good enough. Over time, these weights are fine-tuned to reach higher accuracy by leveraging a small learning rate.

In [ ]:
```
def create_rate_scheduler(train_dataloader: DataLoader, optimizer: Optimizer, epochs = 4):
  """
  Create a learning rate scheduler used with an optimizer
  """
  # Total number of training steps is number of batches * number of epochs.
  total_steps = epochs * len(train_dataloader)

  # Create and return the learning rate scheduler.
  return get_linear_schedule_with_warmup(optimizer, num_warmup_steps = 0, num_training_steps = total_steps)
```

4.4. Creating all Model components

In [ ]:
```
def initialize_model_components(train_dataloader: DataLoader, model_name: str, model_type: str, epochs = 4):
  """
  Initialise the model, optimizer and scheduler, given its necessery parameters
  """
  # Create BERT-based model
  model = create_model(model_type = model_type, model_name = model_name, freeze_model = False)
  model.to(device) # put this model on device

  # Craete AdamW Optimizer
  optimizer = create_AdamW_optimizer(model, lr = 2e-5, eps = 1e-8)

  # Craete Linear Rate Scheduler
  scheduler = create_rate_scheduler(train_dataloader, optimizer, epochs)

  return model, optimizer, scheduler
```

4.5. Setting Random Seed

For reproducability and consistency a random seed is set in used libraries

In [ ]:
```
def set_random_seed(seed_value = 42):
  """
  Setting a random seed in multiple libraries for repoducibility purpose
  """
  random.seed(seed_value)
  np.random.seed(seed_value)
  torch.manual_seed(seed_value)
  torch.cuda.manual_seed_all(seed_value)
```

5. Training-Loop

BERT-like classifiers use a training loop following this steps:

Training:

- Unpack our data from the dataloader and load the data onto the GPU for acceleration
- Clear out gradients calculated in the previous pass
- Perform a forward pass to compute logits and loss
- Perform a backward pass to compute gradients (backpropagation)
- Update the model's parameters (optimizer.step())
- Update the learning rate (scheduler.step())

Evaluation:

- Unpack our data and load onto the GPU
- Forward pass
- Compute loss on validation data and track variables for monitoring progress
- Compute other metrics such as F1 score, precision and recall

In [ ]:
```
def train(model, train_dataloader, validation_dataloader, optimizer, scheduler, epochs,
          do_evaluation = False, loss_function = nn.CrossEntropyLoss()):
  """
  Train the BERT-like models and evaluate if specified
  """
  training_stats = [] # to store values such as training and validaiton loss, accuracy and timings

  print("Start training...\n")
  for epoch_i in range(epochs):

    # =======================================
    #               Training
    # =======================================
    # Print the header of the result table
    print(f"{'Epoch':^7} | {'Batch':^7} | {'Train Loss':^12} | {'Val Loss':^10} | {'Val Acc':^9} | {'F1 Score':^1
    print("-"*107)

    t0_epoch, t0_batch = time.time(), time.time() # Measure the elapsed time of each epoch

    # Reset tracking variables at the beginning of each epoch
    total_loss, batch_loss, batch_counts = 0, 0, 0

    model.train() # Put the model in the training model

    for step, batch in enumerate(train_dataloader):
      batch_counts += 1

      # Uncpack this training batch from the dataloader and load it to GPU
      b_input_ids = batch[0].to(device)
      b_attn_mask = batch[1].to(device)
      b_labels = batch[2].to(device)

      model.zero_grad() # Clear out any previously calculated gradients

      logits = model(b_input_ids, b_attn_mask) # Perform a forward pass. This will return logits.

      # Compute loss and accumulate the loss values
      loss = loss_function(logits, b_labels)
      batch_loss += loss.item()
      total_loss += loss.item()

      loss.backward() # Perform a backward pass to calculate gradients

      # Clip the norm of the gradients to 1.0 to prevent "exploding gradients"
      torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

      # Update parameters and the learning rate
      optimizer.step()
      scheduler.step()

      # Print the loss values and time elapsed for every 20 batches
      if (step % 100 == 0 and step != 0) or (step == len(train_dataloader) - 1):

          time_elapsed = time.time() - t0_batch # Elapsed time for 20 batches

          print(f"{epoch_i + 1:^7} | {step:^7} | {batch_loss / batch_counts:^12.6f} | {'-':^10} | {'-':^9} | {'-'

          # Reset batch tracking variables
          batch_loss, batch_counts = 0, 0
          t0_batch = time.time()
```

```
    # Calculate the average loss over the entire training data
    avg_train_loss = total_loss / len(train_dataloader)

    print('-'*107)
    # ========================================
    #               Evaluation
    # ========================================
    if do_evaluation == True:

      val_loss, val_accuracy, f1_score, recall, precision = evaluate(model, validation_dataloader, loss_function)

      time_elapsed = time.time() - t0_epoch

      # Print performance over the entire training data
      print(f"{epoch_i + 1:^7} | {'-':^7} | {avg_train_loss:^12.6f} | {val_loss:^10.6f} | {val_accuracy:^9.2f} |
      print("-"*107)
    print("\n")

  print("Training complete!")


def evaluate(model, validation_dataloader, loss_function):
    """
    Measure model's performance on validaiton set after each training epoch is done.
    """
    model.eval() # Put the model in evaluation mode. Dropout layers are disabled during evaluation.

    # Tracking variables
    total_val_accuracy = 0
    total_val_loss = 0

    all_labels = []
    all_predictions = []

    for batch in validation_dataloader:
      # Uncpack this training batch from the dataloader and load it to GPU
      b_input_ids = batch[0].to(device)
      b_attn_mask = batch[1].to(device)
      b_labels = batch[2].to(device)

      # Compute logits
      with torch.no_grad():
        logits = model(b_input_ids, b_attn_mask)

       # Compute loss
      loss = loss_function(logits, b_labels)
      total_val_loss += loss.item()

      preds = torch.argmax(logits, dim=1).flatten() # Get the predictions

      # Collect all labels and predictions to calculate precision/recall
      all_labels.append(b_labels.cpu().data.numpy())
      all_predictions.append(preds.cpu().data.numpy())

    predictions = np.concatenate(all_predictions)
    labels = np.concatenate(all_labels)

    # Calculating metrics
    avg_val_loss = total_val_loss / len(validation_dataloader)
    accuracy = accuracy_score(labels, predictions)
    f1 = f1_score(labels, predictions)
    recall = recall_score(labels, predictions)
    precision = precision_score(labels, predictions)

    return avg_val_loss, accuracy, f1, recall, precision
```

6. Testing

The testing loop and evaluation results are implemented in this section

6.1. Testing-Loop

Testing loop is very similar to the evaluation part in the Training Loop. It is done on an independent dataset, does not require epochs, an optimizer and the learning rate scheduler (as it does not backpropagate and learn)

In [ ]:
```python
def testing(model, test_dataloader):
  """
  Test the trained model using test dataloaders
  and return the predictions
  """
  model.eval() # Put the model in evaluation mode

  probabilities, predictions, true_labels = [], [], []
  for batch in test_dataloader:
    # Add batch to GPU
    batch = tuple(t.to(device) for t in batch)

    # Unpack the inputs from our dataloader
    b_input_ids, b_attn_mask, b_labels = batch

    # Compute logits however without backpropagation
    with torch.no_grad():
        logits = model(b_input_ids, b_attn_mask)

    # changing log softmax output to probabilities
    pred_probabilities = torch.exp(logits).detach().cpu().data.numpy()
    preds = torch.argmax(logits, dim=1).flatten() # Get the predictions

    probabilities.append(pred_probabilities)
    predictions.append(preds.cpu().data.numpy())
    true_labels.append(b_labels.cpu().data.numpy())

  all_predictions = np.concatenate(predictions)
  all_labels = np.concatenate(true_labels)
  all_true_probabilities = np.concatenate(probabilities)[:, 1]

  return all_predictions, all_labels, all_true_probabilities
```

6.2. Evaluation Metrics

Evaluate the results of testing loop using common Machine Learning metric like accuracy, recall, precision and f1 score. Moreover, confusion matrix and ROC curve pictures are generated.

In [ ]:
```python
def print_metrics(predictions: np.array, labels: np.array, csv_filename = 'metrics.csv'):
  """
  Print all the metrics and save them in the csv file
  """
  print('Testing results: ')
  print('- Accuracy: ', accuracy_score(labels, predictions))
  print('- F1 Score: ', f1_score(labels, predictions))
  print('- Recall: ', recall_score(labels, predictions))
  print('- Precision: ', precision_score(labels, predictions))
  data = [[accuracy_score(labels, predictions), f1_score(labels, predictions), recall_score(labels, predictions),
  df = pd.DataFrame(data, columns = ['accuracy', 'f1', 'recall', 'precision'])
  df.to_csv(csv_filename)


def print_confusion_matrix(predictions: np.array, labels: np.array, conf_filename = 'confusion_matrix.png'):
  """
  Show the confusion matrix given the data and save it as an image.
  """
  ConfusionMatrixDisplay.from_predictions(labels, predictions)
  plt.savefig(conf_filename)
  plt.show()


def plot_roc_curve(labels: np.array, true_probabilities: np.array, roc_filname = 'ROC.png'):
  """
  Show the ROC curve and save it as an image
  """
  fpr, tpr, thresholds = roc_curve(labels, true_probabilities, pos_label=1)
  # roc curve for tpr = fpr
  random_probs = [0 for i in range(len(labels))]
  p_fpr, p_tpr, _ = roc_curve(labels, random_probs, pos_label=1)

  # plot roc curves
  plt.style.use('seaborn')
  plt.plot(fpr, tpr, linestyle='--',color='orange', label='NN')
  plt.plot(p_fpr, p_tpr, linestyle='--', color='blue')
  plt.title('ROC curve')
  plt.xlabel('False Positive Rate')
  plt.ylabel('True Positive rate')
  plt.legend(loc='best')
  plt.savefig(roc_filname, dpi=300)
  plt.show()


def evaluate_testing_results(predictions: np.array, labels: np.array, true_probabilities: np.array, filename: str
  """
  Evaluate testing results, given the testing-loop results
  """
  csv_filename = filename + '_metrics.csv'
  roc_filename = filename + '_roc.png'
  conf_filename = filename + '_confusion_matrix.png'
  print_metrics(predictions, labels, csv_filename)
  print_confusion_matrix(predictions, labels, conf_filename)
  plot_roc_curve(labels, true_probabilities, roc_filename)
```

6.3. Predict a single claim

Predict a single claim to be either true or false with a given probability

In [ ]:
```python
def model_predict(model, claim):
  """
  Single prediction
  """
  input_id, attention_mask = data_tokenization(np.array([claim]), tokenizer, 80)

  model.eval()

  # Move all components to the same device (gpu or cpu)
  input_id = input_id.to(device)
  attention_mask = attention_mask.to(device)
  model.to(device)

  logits = model(input_id, attention_mask)

  # changing log softmax output to probabilitiesv
  prediction_prob = torch.exp(logits).detach().cpu().numpy()

  return prediction_prob
```

7. Defining Hyperparameters

For defining number of Epochs for later experimentations in every BERT-like model I will choose:

- Batch size: 32 (This is already a batch size of dataloaders)
- Learning rate: 2e-5
- Optimizer: AdamW (The newer version of Adam)
- Epochs: 4 - and decide where to stop
- Max Input Length = ?

And see the point where the data will start to overfit on bert-base model. When the validation loss will start to go up it means that the overfitting is happening and we should not generally continue

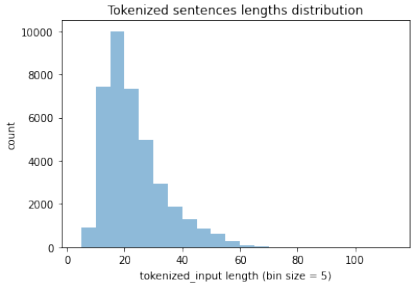
In [ ]:
```
BATCH_SIZE = 32
LEARING_RATE = 2e-5
```

7.1 Deciding on max input length

Seeing what should be the max lengtn of the token size in my BERT model. Tha maximum allowed is 512, however since my dataset is mostly concerend with shorter claims, manual graph insepction is done to determine what should be the max length.

In [ ]:
```
whole_dataset = read_data(path = 'dataset.csv')
tokenizer = load_model_tokenizer(model_name = 'bert-base-cased', model_type = 'bert')
plot_tokenized_data(whole_dataset, tokenizer)
```

```
Reading the dataset...

The data of size  (38659, 2)  has been loaded.
```




```
min:  3 max:  114
```

Altough the maximum tokenized input is 114 we can see that it's an outlier in our dataset. Therefore we will set maximum length to be 80.

In [ ]:
```
MAX_CLAIM_LEN = 80
```

7.2. Deciding on Number of Epochs

The number of epochs is the number of complete passes through the training dataset. To determine the optimal number o epochs for this project, bert-base-cased model is trained and validated for 4 epochs. This is to see where does the validation loss goes up, meaning the model overfits the data.

7.2.1 Preparing data for training

Complete data preprocessing in these steps:

1. Loading training dataset
2. Creating a Tokenizer and tokenize data
3. Splitting data into training and validation
4. Create DataLoaders

In [ ]:
```
# 1) Loading training dataset
print()
print("------------------------------")
print("1.1. Loading training data")
print("------------------------------")
training_df = read_data(path = 'training.csv')
if not training_df.empty:
  # Moving claims and labels to numpy arrays for easier preprocessing
  claims, labels = get_data_as_arrays(training_df)

# 2) Creating a tokenizer
print()
print("------------------------------")
print("1.2. Tokenizing data")
print("------------------------------")
tokenizer = load_model_tokenizer('bert-base-cased', 'bert')
input_ids, attention_masks = data_tokenization(claims, tokenizer, MAX_CLAIM_LEN)
labels = torch.tensor(labels)
print('Tokenization done')


# 3) Splitting data into training and validation
print()
print("------------------------------")
print("1.3. Splitting data")
print("------------------------------")
train_data, validation_data = create_training_validation_data(input_ids, attention_masks, labels, train_percentag

# 4) Create DataLoaders
print()
print("------------------------------")
print("1.4. Create DataLoaders")
print("------------------------------")
train_dataloader = create_dataloader(train_data, isRandomSampler = True, batch_size = BATCH_SIZE)
validation_dataloader = create_dataloader(validation_data, isRandomSampler = False, batch_size = BATCH_SIZE)
print("DataLoaders created")
```

```
------------------------------
1.1. Loading training data
------------------------------
Reading the dataset...

The data of size  (30925, 2)  has been loaded.

------------------------------
1.2. Tokenizing data
------------------------------
Tokenization done

------------------------------
1.3. Splitting data
------------------------------
26,286 training samples
4,639 validation samples

------------------------------
1.4. Create DataLoaders
------------------------------
DataLoaders created
```

7.2.2. Initialise all model components

In [ ]:
```
set_random_seed(42) # Set a random seed for reproducibility purposes
model, optimizer, scheduler = initialize_model_components(train_dataloader, model_name='bert-base-cased', model_t
```

7.2.3. Train BERT model with evaluation

In [ ]:
```
train(
    model,
    train_dataloader,
    validation_dataloader,
    optimizer,
    scheduler,
    epochs = 4,
    do_evaluation = True
)
```

```
Start training...
```

Epoch	Batch	Train Loss	Val Loss	Val Acc	F1 Score	Recall	Precision	Elapsed
1	100	0.606469	-	-	-	-	-	84.50
1	200	0.478827	-	-	-	-	-	84.53
1	300	0.436527	-	-	-	-	-	84.61
1	400	0.418181	-	-	-	-	-	83.76
1	500	0.405173	-	-	-	-	-	83.50
1	600	0.394195	-	-	-	-	-	83.56
1	700	0.389914	-	-	-	-	-	84.55
1	800	0.370411	-	-	-	-	-	84.48
1	821	0.360561	-	-	-	-	-	17.40
1	-	0.435703	0.365687	0.82	0.80	0.78	0.83	734.80

Epoch	Batch	Train Loss	Val Loss	Val Acc	F1 Score	Recall	Precision	Elapsed
2	100	0.315010	-	-	-	-	-	85.41
2	200	0.313183	-	-	-	-	-	84.09
2	300	0.316902	-	-	-	-	-	84.42
2	400	0.317325	-	-	-	-	-	84.31
2	500	0.301759	-	-	-	-	-	84.56
2	600	0.298943	-	-	-	-	-	84.60
2	700	0.298122	-	-	-	-	-	84.53
2	800	0.310236	-	-	-	-	-	84.56
2	821	0.300656	-	-	-	-	-	17.34
2	-	0.308731	0.357660	0.82	0.81	0.79	0.82	737.97

Epoch	Batch	Train Loss	Val Loss	Val Acc	F1 Score	Recall	Precision	Elapsed
3	100	0.233563	-	-	-	-	-	85.30
3	200	0.229016	-	-	-	-	-	84.55
3	300	0.219049	-	-	-	-	-	84.77
3	400	0.214130	-	-	-	-	-	84.61
3	500	0.220635	-	-	-	-	-	84.74
3	600	0.225249	-	-	-	-	-	84.57
3	700	0.214725	-	-	-	-	-	84.43
3	800	0.217596	-	-	-	-	-	84.41
3	821	0.214813	-	-	-	-	-	17.31
3	-	0.221583	0.408851	0.82	0.81	0.84	0.79	738.81

Epoch	Batch	Train Loss	Val Loss	Val Acc	F1 Score	Recall	Precision	Elapsed
4	100	0.173720	-	-	-	-	-	85.28
4	200	0.153944	-	-	-	-	-	84.71
4	300	0.148146	-	-	-	-	-	84.52
4	400	0.162740	-	-	-	-	-	84.34
4	500	0.151753	-	-	-	-	-	84.38
4	600	0.158083	-	-	-	-	-	84.44
4	700	0.147338	-	-	-	-	-	84.40
4	800	0.136622	-	-	-	-	-	84.37
4	821	0.147813	-	-	-	-	-	17.35
4	-	0.153908	0.486927	0.82	0.81	0.83	0.80	737.92

```
Training complete!
```

The ouput shows that the validation loss goes down between epoch 2 and 1 (from 0.365687 to 0.357660) suggesting that the model is learning. However between epoch 3 and 2 the validation loss goes up (from 0.357660 to 0.408851), which means that the model starts to overfit. Hence, EPOCH = 2 is chosen in the further experimantations.

In [ ]:
```
BATCH_SIZE = 32
LEARING_RATE = 2e-5
MAX_CLAIM_LEN = 80
EPOCHS = 2
```

8. BERT Experiments

This section trains and evaluates multiple different BERT-based models, in order to find the best performing one.

Disclaimer: The experiment loop goes through 8 BERT-based models listed in array 'models' and makes use of Google Colab's free GPU. The experimentation were run using the premium subscription of Google Colab. Hence, the GPU is likely to run out of memory on the free version

In [ ]:
```
print("Start training different BERT models")

models = [['bert-base-uncased', 'bert'], ['bert-base-cased', 'bert'], ['bert-large-cased', 'bert'],
          ['bert-large-uncased', 'bert'], ['roberta-base', 'roberta'], ['roberta-large', 'roberta'],
          ['vinai/bertweet-base', 'bertweet'], ['albert-base-v2', 'albert']]

# In order to see the functionality of this part this array can be used instead
# of the original one. The original one will take too much time and GPU power.
# models = [['bert-base-uncased', 'bert'], ['roberta-base', 'roberta']]

print("Loading training and testing datasets: ")

training_df = read_data(path = 'training.csv')
train_claims, train_labels = get_data_as_arrays(training_df)

testing_df = read_data(path = 'testing.csv')
test_claims, test_labels = get_data_as_arrays(testing_df)

for model_array in models:
    model_name = model_array[0]
    model_type = model_array[1]
    print()
    print('-' * 100)
    print(model_name)
    print('-' * 100)
    print()

    # Tokenization
    tokenizer = load_model_tokenizer(model_name, model_type)
    input_ids, attention_masks = data_tokenization(train_claims, tokenizer, MAX_CLAIM_LEN)
    labels = torch.tensor(train_labels)
    dataset = TensorDataset(input_ids, attention_masks, labels)

    # Initialising training components
    train_dataloader = DataLoader(dataset, sampler = RandomSampler(dataset), batch_size = BATCH_SIZE)
    classifier, optimizer, scheduler = initialize_model_components(train_dataloader, model_name, model_type, epoc

    # Training
    set_random_seed(42)

    train(
        model = classifier,
        train_dataloader = train_dataloader,
        validation_dataloader = None,
        optimizer = optimizer,
        scheduler = scheduler,
        epochs = EPOCHS,
        do_evaluation = False
    )

    # Testing
    print("Start testing")
    test_input_ids, test_attention_masks = data_tokenization(test_claims, tokenizer, MAX_CLAIM_LEN)
    labels = torch.tensor(test_labels)
    test_dataset = TensorDataset(test_input_ids, test_attention_masks, labels)

    test_dataloader = DataLoader(test_dataset, sampler = SequentialSampler(test_dataset), batch_size = BATCH_SIZE

    predictions, true_labels, true_probabilities = testing(classifier, test_dataloader)
    print("Testing complete!")

    # Evaluation + Saving the results
    print("Evaluation:")
    evaluate_testing_results(predictions, true_labels, true_probabilities, model_name)

    print('-' * 100)
```

9. Neural Network Experiments

9.1. Roberta For Experiments with different Neural Networks

Special RobertaForExperiments class which allows to pick a preffered neural network (classifier) architecture

In [ ]:
```
class RobertaForExperiments(nn.Module):

  def __init__(self, freeze_model = False, model_name = 'roberta-base', classifier_number = 1):
    super(RobertaForExperiments, self).__init__()
    self.roberta = RobertaModel.from_pretrained(model_name)
    input_size = self.roberta.config.to_dict()['hidden_size']

    # Pick different type of Neural Network
    if classifier_number == 1:
      self.classifier = nn.Sequential(nn.Dropout(0.2), nn.Linear(input_size, 50),
                                      nn.ReLU(), nn.Linear(50, 2), nn.LogSoftmax(dim=1))
    elif classifier_number == 2:
      self.classifier = nn.Sequential(nn.Dropout(0.2), nn.Linear(input_size, 50),
                                      nn.LeakyReLU(), nn.Linear(50, 2), nn.LogSoftmax(dim=1))
    elif classifier_number == 3:
      self.classifier = nn.Sequential(nn.Dropout(0.2), nn.Linear(input_size, 256),
                                      nn.ReLU(), nn.Linear(256, 2), nn.Softmax(dim=1))
    elif classifier_number == 4:
      self.classifier = nn.Sequential(nn.Dropout(0.2), nn.Linear(input_size, 50),
                                      nn.GELU(), nn.Linear(50, 2), nn.LogSoftmax(dim=1))
    elif classifier_number == 5:
      self.classifier = nn.Sequential(nn.Dropout(0.2), nn.Linear(input_size, 50),
                                      nn.SELU(), nn.Linear(50, 2), nn.LogSoftmax(dim=1))

    if freeze_model:
        for param in self.roberta.parameters():
            param.requires_grad = False

  def forward(self, input_ids, attention_mask):
    _, last_cls_token = self.roberta(input_ids = input_ids, attention_mask = attention_mask, return_dict = False)
    logits = self.classifier(last_cls_token)
    return logits
```

9.2. Create Models

Create RobertaForExperiments models with several different neural network classifiers for experimentations

In [ ]:
```
model_1 = RobertaForExperiments(freeze_model = False, model_name = 'roberta-base', classifier_number = 1)
model_2 = RobertaForExperiments(freeze_model = False, model_name = 'roberta-base', classifier_number = 2)
model_3 = RobertaForExperiments(freeze_model = False, model_name = 'roberta-base', classifier_number = 3)
model_4 = RobertaForExperiments(freeze_model = False, model_name = 'roberta-base', classifier_number = 4)
model_5 = RobertaForExperiments(freeze_model = False, model_name = 'roberta-base', classifier_number = 5)

# Models array
models = [model_1, model_2, model_3, model_4, model_5]

# Tokenizer
tokenizer = load_model_tokenizer(model_name = 'roberta-base', model_type = 'roberta')

# Create a dataset
training_df = read_data(path = 'training.csv')
train_claims, train_labels = get_data_as_arrays(training_df)

testing_df = read_data(path = 'testing.csv')
test_claims, test_labels = get_data_as_arrays(testing_df)

input_ids, attention_masks = data_tokenization(train_claims, tokenizer, MAX_CLAIM_LEN)
labels = torch.tensor(train_labels)
dataset = TensorDataset(input_ids, attention_masks, labels)
train_dataloader = DataLoader(dataset, sampler = RandomSampler(dataset), batch_size = BATCH_SIZE)
```

9.3. Experiments

An experimentation loop similalr to the one presented in '8. BERT Experiments'. This time if

In [ ]:
```
number = 1
models = [model_1, model_2, model_3, model_4, model_5]

for model in models:
  model_name = 'roberta' + str(number)
  number += 1
  print('-' * 100)
  print(model_name)
  print('-' * 100)
  print()

  model.to(device)
  optimizer = create_AdamW_optimizer(model, lr = LEARING_RATE, eps = 1e-8)
  scheduler = create_rate_scheduler(train_dataloader, optimizer, EPOCHS)

  # Training
  set_random_seed(42)

  train(
      model = model,
      train_dataloader = train_dataloader,
      validation_dataloader = None,
      optimizer = optimizer,
      scheduler = scheduler,
      epochs = EPOCHS,
      do_evaluation = False
  )

  # Testing
  print("Start testing")
  test_input_ids, test_attention_masks = data_tokenization(test_claims, tokenizer, MAX_CLAIM_LEN)
  labels = torch.tensor(test_labels)
  test_dataset = TensorDataset(test_input_ids, test_attention_masks, labels)

  test_dataloader = DataLoader(test_dataset, sampler = SequentialSampler(test_dataset), batch_size = BATCH_SIZE)

  predictions, true_labels, true_probabilities = testing(model, test_dataloader)
  print("Testing complete!")

  # Evaluation + Saving the results
  print("Evaluation:")
  evaluate_testing_results(predictions, true_labels, true_probabilities, model_name)

  print('-' * 100)
```

10. Training the best model on full dataset

It was decided that the picked model would be 'roberta-base' model with neural network architecture with ReLU() activation function in the hidden layer. Because the model is used in the software it needs to be trained on the whole dataset. This section does that and saves it to the mounted google drive.

10.1. Training Loop

In [ ]:
```
# Read the whole dataset
whole_dataset = read_data(path = 'dataset.csv')
train_claims, train_labels = get_data_as_arrays(whole_dataset)

# Prepare data
tokenizer = load_model_tokenizer(model_name = 'roberta-base', model_type = 'roberta')
input_ids, attention_masks = data_tokenization(train_claims, tokenizer, MAX_CLAIM_LEN)
labels = torch.tensor(train_labels)
dataset = TensorDataset(input_ids, attention_masks, labels)
train_dataloader = DataLoader(dataset, sampler = RandomSampler(dataset), batch_size = BATCH_SIZE)

# Create model components
classifier, optimizer, scheduler = initialize_model_components(train_dataloader, 'roberta-base', 'roberta', epoch

# Train the model
set_random_seed(42)
train(
    model = classifier,
    train_dataloader = train_dataloader,
    validation_dataloader = None,
    optimizer = optimizer,
    scheduler = scheduler,
    epochs = EPOCHS,
    do_evaluation = False
)
```

10.2. Saving the model

In [ ]:
```
# In order to save the model a google drive needs to be mounted
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

In [ ]:
```
# Saving the model on the google drive
path = "/content/gdrive/MyDrive/roberta-base-final.pt"
torch.save(model.state_dict(), path)
```

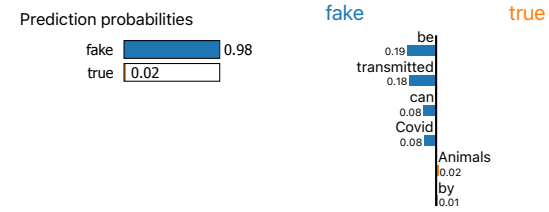
11. LIME Explinability

This part showcases the ability to apply LIME explainability on top of trained RoBERTa model.

In [ ]:
```
# Load trained model from Google Drive
path = "/content/gdrive/MyDrive/roberta-base-final.pt"
model = create_model(model_type = 'roberta', model_name = 'roberta-base', freeze_model = False)
model.load_state_dict(torch.load(path))
model.to(device)
```

In [ ]:
```
def lime_predictor(claims):
  # How to add 2 parameters here -> claims and model
  input_ids, attention_masks = data_tokenization(np.asarray(claims), tokenizer, 80)

  batch_size = 20

  # Create the DataLoader.
  prediction_data = TensorDataset(input_ids, attention_masks)
  prediction_sampler = SequentialSampler(prediction_data)
  prediction_dataloader = DataLoader(prediction_data, sampler=prediction_sampler, batch_size=batch_size)

  model.eval() # This needs to be a model but I can't put

  probabilities = []
  for batch in prediction_dataloader:
    batch = tuple(t.to(device) for t in batch)
    b_input_ids, b_attn_mask = batch

    with torch.no_grad():
      logits = model(b_input_ids, b_attn_mask)

      pred_probabilities = torch.exp(logits).detach().cpu().data.numpy()
      probabilities.append(pred_probabilities)

  return np.vstack(probabilities) # array of probabilities for each sample
```

In [ ]:
```
explainer = LimeTextExplainer(class_names=['fake', 'true'])
str_to_predict = "Covid can be transmitted by Animals"
exp = explainer.explain_instance(str_to_predict, lime_predictor, num_features=10, num_samples=1000)
exp.show_in_notebook(text=str_to_predict)
```



Text with highlighted words

Covid can be transmitted by Animals

C.4 Streamlit Website Code

C.4.1 app.py

```
"""
Main file of the application that gets called at the beginning of each session.
It specifies necessery configurations and components.
"""

import streamlit as st
from layout import sidebar


def define_page_config():
    """
    Set overall configuration for the app. Specify the layout, sidebar type etc
    """
    # st.set_page_config(layout="wide", initial_sidebar_state = 'collapsed', page_title='Fake Take', page_icon="â")

    st.set_page_config(layout="wide", page_title='Fake Take', page_icon="â")
    # Specify the main container of the website and its padding
    st.markdown(
        f"""
        <style>
            .appview-container .main .block-container{{
                padding-top: {0}rem;
                padding-right: {12}%;
                padding-left: {12}%;
                padding-bottom: {0}rem;
            }}
        </style>
        """,
            unsafe_allow_html=True,
        )

def local_css(file_name: str):
    """
    Define local css file
    """
    with open(file_name) as f:
        st.markdown(f'<style>{f.read()}</style>', unsafe_allow_html=True)


def main():
    """
    Main method
    """
    define_page_config() # define page config
    local_css("./assets/style.css") # define local css
    sidebar() # define page navigation sidebar


if __name__ == '__main__':
    main()
```

C.4.2 layout.py

```
"""
Main UI file for displaying content on the website.
It creates and generates necessery components for  "home"
page and "about" page accessible from the sidebar
"""

import streamlit as st
from streamlit_option_menu import option_menu
import streamlit.components.v1 as components
import webbrowser
from model import load_model_components, model_predict
from lime_explainer import lime_explanation


def display_logo_components():
    """
    Display logo components for the main home page.
    """
    _, col2, _ = st.columns([3,6,3])
    with col2:
        # Logo + text
        st.image('./assets/logo.png' , use_column_width = True)
        st.markdown('<h5 style="text-align:center;">An attempt to combat misinformation online</h5>',unsafe_allow_html=True)

        # Display loading message if the model and tokenizer are not yet loaded into this session
        if 'model_initialised' not in st.session_state or 'model_initialised' not in st.session_state:
            gif_runner = st.image('./assets/loading.gif')
            text_runner = st.markdown('<p style="text-align:center;">Loading necessary componenets...
</p>',unsafe_allow_html=True)

            # Load model components and record this in session state
            load_model_components()
            st.session_state['model_initialised'] = True
            st.session_state['tokenizer_initialised'] = True

            gif_runner.empty()
            text_runner.empty()


def display_lime_explanation(input_claim: str):
    """
    Display LIME explanation
    """
    explanation = lime_explanation(input_claim)

    explanation_placeholder = st.empty()
    placeholder = st.empty()
    with explanation_placeholder:
        with st.expander("See output explanation", expanded = False):
            st.write("""
                This fake news detection model outputs weather it predicts the claim to be true or false and gives its probability
estimation.
                Fruthermore it gives insight into how particular words influenced model's decision and lean more towards fake/true
sentiment.
                For more details please see "About" page.

                ***Disclaimer: given output is only an estimate. Use trusted sources to validate news.***
            """)
    with placeholder:
        components.html(explanation, height = 400, scrolling = True)


def display_prediction_result(input_claim: str):
    """
    Display prediction results and corresponding graphics
    """
```

```
    prediction, prediction_prob = model_predict(input_claim) # Get the prediction and its probability
    if prediction == True:
        emoji = 'â'
    else:
        emoji = 'â'

    st.write("")
    st.markdown(f"""
        <div style="display: flex; justify-content: center; text-align: center; font-size:30-px; width: 100%;">
            <h5>{emoji}  This claim is considered to be {prediction} with the probability of {str(round(prediction_prob* 100, 1))}%
{emoji} </h5>
        </div>
            """, unsafe_allow_html=True)
    st.write("")


def sidebar():
    """
    Create a navigation sidebar
    """
    with st.sidebar:
        choose = option_menu("", ["Home", "About"],
            icons=['house', 'info-circle'],
            menu_icon="app-indicator", default_index=0,
            styles={
                "container": {"padding": "5!important", "background": "rgba(0,0,0,0)"},
                "icon": {"color": "green", "font-size": "30px"},
                "nav-link": {"font-size": "25px", "text-align": "left", "margin":"0px"},
                "nav-link-selected": {"background": "rgba(0,0,0,0)", "color": 'black'},
            }
        )
    if choose == 'Home':
        homePage()
    elif choose == 'About':
        aboutPage()


def homePage():
    """
    Create home page components
    """
    display_logo_components()

    # Claim to predict veracity
    input_claim = st.text_input("Enter the claim that you want to fact-check")

    if st.button("Validate"):
        if len(input_claim) != 0: # Proceed only if there is some text

            # Get prediction and its graphics
            display_prediction_result(input_claim)

            # Display the gif and a loading text
            gif_runner = st.image('./assets/loading.gif')
            text_runner = st.markdown('<p style="text-align:center;">Generating Explanation...</p>',unsafe_allow_html=True)

            # Get LIME explanation
            display_lime_explanation(input_claim)

            # Clear the gif and loading text
            gif_runner.empty()
            text_runner.empty()

        elif len(input_claim) == 0:
            st.warning("Input a claim you want to fact check above")
```

```
    st.markdown('<p style="text-align:center;">Fake Take is just an automated classifier, make sure to use trusted sources for fact-checking news.</p>',unsafe_allow_html=True)



def aboutPage():
    """
    Create About page components
    """
    # About title
    st.write("###")
    st.markdown('<h5 style="font-size: 40px;">About Fake Take</h5>',unsafe_allow_html=True)

    # Description
    st.write("""
        Fake Take is an easy to use web tool for inspecting the results of undergraduate project:
        ***A thorough attempt to enhance fake news detection through unbiased dataset, explainability and BERT-based models*** made by ***Jan Marczak***

        It serves as a place to test and experiment with state of the art deep-learning BERT-base machine
        learning model for natural language tasks called **RoBERTa**, that was pre trained on lots of data to
        be able to understand english language. The model was then additionally trained on carefully constructed
        dataset that combines multiple fake news detection datasets into one, in a thoughtful manner.

        To get more insight into model's black-box-like execution, **LIME** AI Explainability method was applied
        on top, to see how each word influenced model's decision. Red color indicates word's tendency towards
        fake prediction and vice-versa with green as can be seen on this example:
    """)

    # Lime example
    _, col2, _ = st.columns([3,8,3])
    with col2:
        st.image('./assets/lime_example.png')

    st.write("""
        To learn more about it you can visit its github page linked below or download the project report paper.
        For any questions or suggestions contact jan.marczak00@gmail.com
    """)

    # GitHub and download report buttons
    col1, col2 = st.columns([1]*1+[1.1])
    with col1:
        if st.button('Visit GitHub'):
            webbrowser.open_new_tab('https://github.com/janmarczak')
    with col2:
        with open("./assets/bspr.pdf", "rb") as pdf_file:
            PDFbyte = pdf_file.read()

            st.download_button(label="Download Project Report",
                               data=PDFbyte,
                               file_name="./assets/bspr.pdf",
                               mime='application/octet-stream')
```

C.4.3 model.py

```
"""
File containing all model components and functions.
It defines the architecture class for RoBERTa model
and creates necessery components like RoBERTa tokenizer
"""

from numpy import array
from transformers import RobertaModel, RobertaTokenizer
import torch.nn as nn
import torch
from transformers import logging
logging.set_verbosity_error()


class RobertaForFakeNewsDetection(nn.Module):
  """
  RoBERTa Model for classyfing claims to true/false.
  """

  def __init__(self):
    """
    @param    classifier: a torch.nn.Module classifier
    @param    freeze_bert (bool): Set 'False' to fine-tune the BERT model
    """
    super(RobertaForFakeNewsDetection, self).__init__()
    # Instantiate BERT model
    self.roberta = RobertaModel.from_pretrained('roberta-base')
    # Our classifier (feed forward neural network)
    self.classifier = nn.Sequential(
        nn.Dropout(0.2),            # Dropout regulatization method (to avoid overfitting)
        nn.Linear(768, 50),         # Hidden Layer
        nn.ReLU(),                  # Activation Function
        nn.Linear(50, 2),           # Output Layer
        nn.LogSoftmax(dim=1)        # Activation Function in the output
    )


  def forward(self, input_ids, attention_mask):
    """
    Feed input to BERT and the classifier to compute logits.
    @param    input_ids (torch.Tensor): an input tensor with shape (batch_size, max_length)
    @param    attention_mask (torch.Tensor): a tensor that hold attention mask information with shape (batch_size, max_length)
    @return   logits (torch.Tensor): an output tensor with shape (batch_size, num_labels)
    """

    # Feed input to bert and extract last hidden state of the [CLS] token, that is used for classificaiton task.
    _, last_cls_token = self.roberta(input_ids = input_ids, attention_mask = attention_mask, return_dict = False)

    # Feed input to classifier to compute logits ()
    logits = self.classifier(last_cls_token)

    return logits


def data_tokenization(claims: list, max_claim_length = 80):
    """
    Tokenize data with RoBERTa tokenizer
    """
    data_tokens = tokenizer.__call__(
        claims,                         # Sentence to encode.
        add_special_tokens = True,      # Add '[CLS]' and '[SEP]'.
        padding = True,                 # Pad sentences.
        truncation = True,              # Truncate sentences.
        max_length = max_claim_length,  # Specify max length
        return_attention_mask = True,   # Construct attn. masks.
    )
```

```
    # Convert the lists into tensors.
    input_ids = torch.tensor(data_tokens['input_ids'])
    attention_masks = torch.tensor(data_tokens['attention_mask']) # differentiates padding from non-padding

    return input_ids, attention_masks


def model_predict(claim: str):
    """
    Predict a single claim with probability
    """
    input_id, attention_mask = data_tokenization([claim], 80)

    classifier.eval()
    logits = classifier(input_id, attention_mask)

    # changing log softmax output to probabilitiesv
    prediction_prob = torch.exp(logits).detach().cpu().numpy()

    if prediction_prob[0][0] > prediction_prob[0][1]:
        prediction = False
    else:
        prediction = True

    return prediction, max(prediction_prob[0])


def load_roberta_model():
    """
    Initialise and load RoBERTa model configuration from the file.
    """
    global classifier
    classifier = RobertaForFakeNewsDetection()
    classifier.load_state_dict(torch.load('./model/roberta-base.pt', map_location=torch.device('cpu')))

def get_classifier():
    """
    Get the classifier
    """
    return classifier

def load_tokenizer():
    """
    Cretae RoBERTa tokenizer
    """
    global tokenizer
    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')

def load_model_components():
    """
    Load RoBERTa model and its tokenizer
    """
    load_roberta_model()
    load_tokenizer()
```

C.4.4 lime_explainer.py

```
"""
File containing functions for generating and formatting LIME (XAI)
explanations after the predictions is made
"""

import lime
from lime.lime_text import LimeTextExplainer
from torch.utils.data import TensorDataset, DataLoader, SequentialSampler
import numpy as np
from model import data_tokenization, get_classifier
import torch


def lime_predictor(claims: np.array):
    """
    Model prediction function that takes a numpy array and ouputs prediction probabilities.
    Used in the process of creating explanations for claims with LIME
    """
    input_ids, attention_masks = data_tokenization(claims, 80)
    batch_size = 20
    device = 'cpu'

    prediction_data = TensorDataset(input_ids, attention_masks)
    prediction_sampler = SequentialSampler(prediction_data)
    prediction_dataloader = DataLoader(prediction_data, sampler=prediction_sampler, batch_size=batch_size)

    classifier = get_classifier()
    classifier.eval()

    probabilities = []
    for batch in prediction_dataloader:
        batch = tuple(t.to(device) for t in batch)
        b_input_ids, b_attn_mask = batch

        with torch.no_grad():
            logits = classifier(b_input_ids, b_attn_mask)

        pred_probabilities = torch.exp(logits).detach().cpu().data.numpy()
        probabilities.append(pred_probabilities)

    return np.vstack(probabilities) # array of probabilities for each sample


def format_lime_html(lime_str: str):
    """
    Format HTML file generated from LIME explanation
    """
    # Edit css files for HTML components
    lime_str = lime_str.replace(r".lime.top_div {\n  display: flex;\n  flex-wrap: wrap;",
                r".lime.top_div {\n  display: flex;\n  flex-direction: row;\n  flex-wrap: wrap;\n  justify-content: space-evenly;\n  height:
auto;")
    lime_str = lime_str.replace(r".lime.predict_proba {\n  width: 245px;",
                r".lime.predict_proba {\n  width: auto;\n  display: inline-block;\n  color:  rgb(250, 31, 31);")
    lime_str = lime_str.replace(r".lime.explanation {\n  width: 350px;",
                r".lime.explanation {\n  width: auto;\n  display: inline-block;")
    lime_str = lime_str.replace(r".lime.text_div {\n  max-height:300px;\n  flex: 1 0 300px;\n  overflow:scroll;",
                r".lime.text_div {\n  width: auto;\n  display: inline-block;")

    # Changing the colors of fake/true predictions to red & green
    lime_str = lime_str.replace("colors = [this.colors_i(0), this.colors_i(1)];", "colors = [this.colors_i(3), this.colors_i(2)];")
    lime_str = lime_str.replace(r", 1, exp_div);", r", 2, exp_div);")
    lime_str = lime_str.replace("var color = this.colors(names[i]);", "var color = this.colors(names[data.length + 1 - i]);")

    return lime_str
```

```
def lime_explanation(claim: str):
    """
    Run process of explaining claim's predictions using LIME
    """
    explainer = LimeTextExplainer(class_names=['fake', 'true'])
    exp = explainer.explain_instance(claim, lime_predictor, num_features=10, num_samples=1000)
    exp_str = exp.as_html()

    return format_lime_html(exp_str)
```