Malicious URL Detection with Large Language Models

Tuesday 25th March, 2025 - 10:31

Ivaylo Krumov
University of Luxembourg
Email: ivaylo.krumov.001@student.uni.lu

This report has been produced under the supervision of:
Salima Lamsiyah
University of Luxembourg
Email: salima.lamsiyah@uni.lu

Abstract

The purpose of this Bachelor Semester Project is to determine the feasibility of large language models in the context of their utilization for detecting malicious URLs. To find this out, the project focuses on the domains of cyber threat intelligence, machine learning and natural language processing. In order to produce concrete results, a number of concepts and principles associated with the project's topic have been explored in detail, establishing the scientific context around which the project is created. Furthermore, several programs representing different implementations of large language models have been fine-tuned and optimized with the aim of showing their potential in distinguishing between benign and malicious URLs. All accomplished results of the project are thoroughly presented in this report. The ultimate goal of the report and the project as a whole is to give useful insight into the ways LLMs and relevant techniques from machine learning and natural language processing can be employed for identifying malicious URLs with the intent to enhance cybersecurity methods of mitigating the impact of these threats.

1. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
  8) Using another person's unpublished work without attribution and permission ('stealing')
  9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

2. Introduction

In today's interconnected world, there exists a rapid exchange of information for both personal and professional activities via the internet. However, as technology advances so do the methods utilized by cybercriminals. In particular, one of the oldest and most common cyberattacks is the use of malicious URLs designed to deceive users and gain unauthorized access to sensitive information. Nowadays there are

many sophisticated ways that cybercriminals employ so that they can perform this effectively, making the detection of such URLs critical in cybersecurity and cyber threat intelligence, as it is aimed at protecting users from threats like phishing and malware. Traditional methods, such as blacklists, have proven less and less effective against the evolving tactics of attackers, leading to a need for more sophisticated approaches involving machine learning and natural language processing (NLP).

This project and report explore the application of large language models (LLMs) for detecting malicious URLs. These advanced neural network architectures use large amounts of textual data to generate responses based on given input and this ability could be utilized in identifying patterns in web address links that are indicative of malicious intent to predict if a URL is benign or not. We examine the principles behind neural networks and transformer-based architectures, their capabilities in pattern recognition and contextual analysis, as well as the importance of data pre-processing, feature extraction, and fine-tuning in enhancing model performance for cybersecurity applications. To provide better understanding in how LLMs can be utilized effectively for this task, we also review relevant case studies and examples from recent research, which show the effectiveness of various machine learning techniques, both traditional and deep learning approaches, in identifying and mitigating cyber threats.

Moreover, an implementation of LLMs fine-tuned for URL classification is introduced to support this exploration. It integrates most of the discussed concepts to provide a clear example of the practical effectiveness of employing LLMs for the use of malicious URL detection. To determine their potential, different metrics are used for evaluating the results that each one generates after fine-tuning them to accommodate for the classification task. In the end, these results are discussed in a comparison with one another in order to find out which of the presented LLMs and respective techniques can most reliably be used in malicious URL detection.

3. Project description

3.1. Domains

3.1.1. Scientific. The project focuses on the following scientific domains:

- **Cybersecurity.** The field of cybersecurity is dedicated to protecting systems, networks, and programs from cyberattacks. Its focus is on preventing unauthorized access to sensitive data and ensuring the integrity and availability of data. In the context of this project, the subfield of cyber threat intelligence is especially emphasized. It specifically targets the topic of preventing of URL-based cyberattacks by detecting of malicious URLs, which are commonly used in phishing attacks and malware distribution.
- **Natural language processing (NLP).** Natural language processing is a branch of artificial intelligence that deals with the analysis and interpretation of human language by computers. It involves the use of algorithms and computational techniques to process written or spoken language. Within this project, NLP techniques are utilized to pre-process URL text data so that it can be further used for training large language models to detect malicious URLs by classifying them as either benign or malicious.
- **Machine learning.** A subfield of artificial intelligence, machine learning is focused on algorithms that enable computers to learn from data and make decisions based on this learning process. In this project, machine learning principles are applied to train large language models for the task of URL classification. This includes understanding several machine learning algorithms and techniques, such as text classification and model fine-tuning. The goal is to optimize these models as much as possible to accurately distinguish between safe and harmful URLs, maximizing the ability to detect and prevent URL-based threats.

3.1.2. Technical. The project encompasses the following technical domains:

- **Python.** Python is the programming language of choice for the technical part of the project due to its high popularity and versatility, as well as ease of use. Its clear high-level syntax makes it accessible for both beginners and experts. Python's large range of libraries support various domains, including natural language processing and machine learning with Scikit-learn, Tensorflow, and PyTorch and data processing with Pandas, Numpy, and Matplotlib. These tools are essential for handling large datasets, building and training models and visualizing results.
- **Large language models (LLMs).** Large language models are a rather recent but significant advancement in both natural language processing and machine learning. LLMs are artificial neural networks that are pre-trained on vast amounts of text data and can be fine-tuned for specific tasks. In this project, LLMs are utilized to recognize and classify URLs based on provided URL data, which involves fine-tuning these models to adapt to the specific task of malicious URL detection and improve their classification accuracy.
- **Data processing.** The application of data processing includes the aspects of pre-processing, analyzing and visualizing data to extract meaningful information about said data. In the context of this project, these skills are necessary for handling the chosen large dataset of URLs. In particular, these are data pre-processing, feature extraction, and visualization of results. Specific tasks are supported by different tools and libraries, such as Pandas and Numpy for data processing and Matplotlib and Seaborn for data visualization.

3.2. Targeted Deliverables

3.2.1. Scientific deliverable. The produced scientific deliverable aims to find an answer to the scientific question "How can

we use large language models to detect malicious URLs?". To accomplish this, it makes use of the information and findings of multiple relevant scientific papers in order to gather the knowledge necessary for an adequate answer to the question at hand. Using these sources as a knowledge base, it discusses topics such as theory and architecture of large language models and current state of cybersecurity and cyber threat intelligence in regard to malicious URL detection. Once the relevant aspects related to the question have been covered, a definitive answer is formulated based on the gained understanding. In the end, there is a brief discussion to assess the extent to which all of the requirements for the deliverable have been satisfied.

3.2.2. Technical deliverable. The produced technical deliverable consists of multiple programs in a Jupyter notebook format, each of which represents a different model trained and fine-tuned for URL classification using the same dataset of URLs. In this report each step of the programs' code is described thoroughly, including pre-processing the URL data, definition of the model, training and fine-tuning. The final results of the validation of each model and their visualizations are then also discussed, serving to support the answer to the scientific question of the project, provided in the scientific deliverable. Finally, there is a short section that assesses to what extent the work done meets the expectations for the deliverable, as well as the improvements that could have been made given more time and resources.

4. Pre-requisites

The complete realization of this project, both the scientific and the technical part, takes advantage of several skills and competencies that were already possessed before the start of the project's research process. For each deliverable, these pre-requisites are respectively described in the 2 subsections that follow.

4.1. Scientific pre-requisites

The author is familiar with the proper writing and organizational techniques for a piece of writing with a scientific theme. Additionally, as long as the terminology used is at a reasonable level of complexity, he is fully capable of understanding meaningful information from scientific sources. Last but not least, he has a general knowledge of the scientific fields involved in the project, namely cybersecurity and cyber threat intelligence, particularly in the practical use of large language models for malicious URL detection. This includes an understanding of natural language processing techniques, text classification, pattern recognition, and familiarity with existing work related to the detection of malicious web addresses.

4.2. Technical pre-requisites

The author has prior experience in programming in several languages. As a result, he is well aware of the possibility of running into technical challenges while developing the technical deliverable and is ready to take the necessary steps to overcome them. The author also has good proficiency with Python programming, is able to make use of a variety of Python packages and is capable of applying his technical skills in classification tasks. This includes a practical understanding of basic machine learning skills (e.g. model training and evaluation), processing and analysis of datasets and basic data visualization skills essential for analysis and interpretation of the results produced at the end of the classification process.

5. Scientific Deliverable

5.1. Requirements

The scientific deliverable of this project aims to satisfy several requirements, which are presented in this section. In order to provide better quality for the structure of their presentation, they are split into two different categories, namely functional and non-functional requirements. The functional requirements are described first, followed by a description of the non-functional ones.

5.1.1. Functional requirements:. The project's scientific deliverable should focus on providing an answer to the scientific question "How can we use large language models to detect malicious URLs?". To that end, it should give explanations of the concepts relevant to the project, such as fine-tuning pre-trained LLMs, data pre-processing, and evaluation of model performance. Additionally, the scientific deliverable should base itself on relevant academic and scientific sources to provide an overview of the field of cybersecurity and cyber threat intelligence, as well as context for the explored techniques and models. The latter includes discussing the underlying principles of machine learning, natural language processing and specific characteristics of model types relevant to the ones considered for the technical deliverable, such as neural networks and models with transformer-based architecture.

5.1.2. Non-functional requirements:. The scientific ideas should be presented in a manner that is fairly straightforward to understand by readers without a background in machine learning, provided that the necessary scientific terms are well defined. The overall structure of the deliverable should ensure that its contents are coherent and logically connected to the main objective of evaluating the potential of LLMs for usage in URL classification. All sections must contribute towards achieving the final goal of providing a detailed description of the concepts relevant to the project's topic and the given scientific question. Additionally, the deliverable should appropriately cite all references whenever they are used, making sure that proper credit is given to the original authors and their work. It should also be structured in such a way that it follows a clear, concise and organized format in order to provide good and easy readability.

5.2. Design

The scientific deliverable is a text that aims to complement the technical deliverable by providing context to its ideas and implementation. While the project's overall success relies more heavily on the results of the technical deliverable, the scientific deliverable remains a crucial component of the process as it gives the basic understanding necessary to grasp the technical work later on. To ensure a comprehensive exploration of the relevant topics, the complete research for the scientific deliverable was conducted over a span of approximately 4 weeks.

The structure of the scientific deliverable is straightforward — it is divided into a series of sections, each dedicated to defining a specific concept or subtopic related to the scientific question. Once all required definitions and explanations are laid out, the text synthesizes this information to formulate a conclusive answer to the scientific question. This concluding section integrates the key points made in the previous sections to present a cohesive and well-founded answer that bridges the gap between the theoretical and practical aspects of the project.

Multiple scientific sources are referenced throughout the scientific deliverable to guarantee that the information presented is accurate and well-founded. These sources provide formal definitions and theoretical concepts and add to the clarity and specificity of the explanations. Wherever applicable, concrete examples are provided to illustrate the concepts in a more detailed way that gives better understanding to the reader.

5.3. Production

In order to answer the scientific question: "How can we use large language models to detect malicious URLs?", we will need to delve deeper into the concepts and principles relevant to the areas that it encompasses in the context of malicious URL detection, namely LLM usage and architecture, machine learning and natural language processing (NLP) techniques and the current state of cyber threat intelligence based on existing case studies. To that end, we will begin the search for an answer to the question by exploring the princpiles behind LLMs and the process of fine-tuning them for application in specific use cases.

Neural networks in general are complex computational models designed to recognize patterns by learning from data through multiple layers of interconnected nodes, also known as neurons. Each neuron processes input data, applies a transformation algorithm, and passes the output to the next layer, allowing the network to learn increasingly complex representations of the data. This hierarchical structure enables neural networks to excel in tasks where understanding complex patterns is a necessity, such as image and text classification. Large language models, such as BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer), are a type of neural network models that have revolutionized the field of NLP by utilizing vast amounts of text data to learn linguistic patterns and contextual information. These models are built on a transformer-based architecture [2], which is a relatively new concept and represents a significant departure from traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs) that were previously dominant in the field of NLP. Transformers consist of an encoder and a decoder, both made up of multiple layers of self-attention mechanisms and feed-forward neural networks. The encoder processes the input data, transforming it into a continuous representation that captures the context of each word or token relative to others in the sequence. The decoder then uses this representation to generate the output, making transformers particularly effective for tasks that require understanding and generating language.

The self-attention mechanism is a crucial part of transformer architectures. Unlike RNNs, which process input sequences sequentially, transformers can process entire sequences in parallel, significantly improving computational efficiency. Self-attention allows the model to weigh the importance of different parts of the input sequence, enabling it to focus on the most relevant features for a given task. This is achieved through the calculation of attention scores, which determine how much influence each word or token should have on the others. In practice, the ability to capture the context within the data makes transformers exceptionally powerful for a wide range of NLP tasks, including text classification, machine translation, and question answering. Another significant advantage of transformers that is derived from this is their scalability, as by leveraging parallel processing and attention mechanisms, transformers can be trained on very large datasets. This results in models that generalize well across various tasks and domains and makes them especially suitable for classification tasks, where it is expected that a large amount of labeled input data must be processed by the model in order to produce desirable outcomes.

Fine-tuning pre-trained transformers, such as BERT and GPT, involves adapting these models to specific tasks by taking advantage of their already extensive knowledge of langiage and training them on domain-specific datasets. During fine-tuning the model is exposed to examples from the target domain, allowing it to learn the specific features and patterns relevant to the task at hand. For malicious URL detection in particular, this would mean training the model on datasets containing both benign and malicious URLs so that it is able to learn the structural characteristics indicative of malicious intent. A critical aspect of fine-tuning is providing meaningful data by first pre-processing it. This means cleaning the data to remove any irrelevant information, while extracting relevant features, and encoding these features into a format suitable for the model. When it comes to URL data, feature extraction includes analyzing the length of the URL, the presence of special characters, the use of IP addresses instead of domain names and the overall structure of the URL [3]. These features are then transformed into numerical representations using techniques such as tokenization and embedding, which convert the raw text data into vectors that the model can process.

The actual fine-tuning process typically consists of several steps. First, the pre-trained model is initialized with weights from its training on a large corpus of general text. Next, the model is further trained on the pre-processed dataset, adjusting its weights to better capture the patterns of the task. This training process includes optimizing the model's parameters using techniques such as the commonly used gradient descent, where the model iteratively improves its performance by minimizing a loss function that numerically describes the difference between its predictions and the actual outcomes. In addition to fine-tuning, transformers benefit from hyperparameter tuning to optimize their performance for a specific task. Hyperparameters include the learning rate, batch size, and the number of training epochs. Usually they are adjusted to find the optimal configuration that maximizes the model's metrics. This tuning process often involves experimenting with different settings and evaluating the model's performance on a validation set (typically extracted from the same dataset) to check if it manages to perform well on unseen data, which should be the case ideally.

As mentioned earlier, fine-tuning is dependent on different concepts and techniques to optimize a model's performance for a specific task. Since the fields of machine learning and NLP collectively encompass too many aspects than we can discuss in this report, we will focus only on those that are actually relevant to the project, namely supervised learning, tokenization and vectorization. URL classification, which is essentially what we are performing when distinguishing between benign and malicious URLs, falls under the concept of supervised learning. This is the most commonly used approach for machine learning and it consists of training a model on a labeled dataset, where each example is associated with a known output. The fine-tuning process of an LLM is generally very similar to its original supervised training, where a selected supervised learning algorithm is used to iteratively adjust the model's parameters so that the difference between its predictions and the actual labels is minimized. As already mentioned, the optimization process relies on a mathematical loss function, which computes the error in the model's predictions. Iteratively minimizing this loss function leads to its value decreasing with each iteration, which is an indication of the model's learning performance and its accuracy in predicting the correct labels. Once the model is trained, it can be evaluated on a separate test set to assess its performance. Common evaluation metrics include accuracy, precision, recall, and F1-score, which provide insights into the model's ability to correctly predict the labels while minimizing false positives and false negatives.

Tokenization and vectorization are two important data preprocessing aspects that are particularly relevant to LLMs. A LLM needs to process input data in a suitable way so that it can effectively learn from it. In the context of URL data, tokenization can break down a URL into smaller components known as tokens, such as domain names, subdomains, paths, and query parameters. Tokenization helps in simplifying the URLs into manageable pieces that the LLM can analyze individually. But a LLM cannot simply take raw text data as input, as it is fundamentally limited to processing only numerical data, which is why vectorization needs to be applied. Through the use of vectorization, the tokens are converted into numerical representations in the form of vectors of numbers that capture their semantic meaning and contextual relationships between each other. There are multiple widely used approaches that are used for the conversion, such as Bag-of-Words, Term Frequency-Inverse Document Frequency and word embeddings, but we will not explore them here, as usually when fine-tuning a LLM the vectorization process is implicit so we would not need detailed knowledge on it.

With these principles covered, we can look into a few existing cases of machine learning model usage in the context of identifying malicious URLs. In the field of cyber threat intelligence, machine learning models, especially LLMs, play a crucial role in detecting and countering cyber threats by analyzing vast amounts of data quickly and accurately. A study by Saleem Raja et al. [3] demonstrated the effectiveness of using lexical features and machine learning classifiers like Random Forest to detect malicious URLs with high accuracy. The study highlighted that features such as URL length and the presence of special characters were significant indicators of malicious intent. By focusing on these features, the model achieved a detection accuracy of 99%. Another study by Abdi and Wenjuan [4] explored the use of convolutional neural networks for detecting malicious URLs. The researchers used a dataset comprising over 420,000 URLs, including both benign and malicious ones. The CNN model was able to learn complex patterns in the data, achieving an accuracy rate of over 96%. Taking the use of CNNs one step further, Hung Le et al. [7] introduced URLNet - a deep learning framework that applies CNNs to both characters and words of the URL string to learn a non-linear URL embedding for malicious URL detection. The study demonstrated that URLNet significantly outperformed traditional machine learning approaches by capturing semantic and sequential patterns within the URL strings. The CNN-based approach was able to achieve high detection rates, even when classifying new URLs it had not seen before. Shifting focus towards LLMs, Verma and Das [5] investigated the use of character N-grams for URL analysis. N-grams are sequences of n successive characters extracted from a text, e.g. a 3-gram of the word "example" would be "exa", "xam", "amp", etc. The authors demonstrated that by analyzing these n-grams in a URL, the model can effectively detect patterns that are typical of malicious URLs, such as uncommon character sequences like multiple dots or hyphens within a short segment which were often flagged as suspicious and this helped in identifying malicious URLs that traditional methods might miss. Their results also show that applying this approach could enhance the efficiency and speed of detection models. A review by Sahoo et al. [6] presented a comprehensive survey of malicious URL detection techniques, emphasizing the role of machine learning and deep learning models in improving detection rates and reducing false positives. An example discussed in the

review talks about the integration of multiple machine learning techniques, including logistic regression and neural networks, to improve the robustness of detection systems. This hybrid approach was shown to reduce false positives significantly while maintaining high detection accuracy, highlighting the potential of combining different models to improve overall performance.

The in-depth theoretical exploration and case studies presented in this deliverable show that LLMs can be well fitted for the detection of malicious URLs. The shown examples, in combination with the presented theoretical knowledge, provide a concrete answer to the scientific question - it is evident that by making use of the principles of neural networks, transformer-based architectures, and supervised learning, LLMs can effectively analyze and identify malicious patterns within URLs. The success of these models is highly dependent on the quality of the data pre-processing, feature extraction, fine-tuning and hyperparameter adjustment processes. As demonstrated by the different studies, integrating different machine learning techniques and continuously refining models through fine-tuning and hyperparameter optimization can further lead to more accurate and efficient detection systems.

5.4. Assessment

The scientific deliverable successfully provides detailed explanations of relevant concepts such as fine-tuning pre-trained LLMs, data pre-processing and evaluation of model performance, establishing a solid foundation for understanding the inner workings of LLMs. Moreover, it contextualizes this theoretical knowledge for the particular use case of malicious URL detection by presenting real-world examples of how NLP models have been implemented to perform this task. This comprehensive exploration successfully addresses the deliverable's scientific question primary and provides a concrete answer to it.

The overall structure of the scientific deliverable is organized so that its contents are clear, coherent and logically connected to the main objective. Each section builds upon the previous one, while comprehensively exploring the topic at hand. Furthermore, the deliverable makes good use of different scientific papers as references throughout to strengthen the credibility of the presented information and give a deeper understanding of the topic. The inclusion of case studies and examples serves as a bridge between the discussed theoretical concepts and real-world applications, as well as the two deliverables of this project.

Overall, the scientific deliverable successfully meets all the expectations that were placed on it. The reader is prepared to explore the technical deliverable by understanding the key concepts included in it and the justification for their utilization in its implementation thanks to the scientific context that has been provided for it.

6. Technical Deliverable

6.1. Requirements

The technical deliverable of this project aims to satisfy several requirements, which are presented in this section. To provide a better structure, they are split into two different categories: functional and non-functional requirements. The functional requirements are described first, followed by a description of the non-functional ones.

6.1.1. Functional requirements. The technical deliverable is expected to include the implementation of at least two large language models tailored for URL classification. The primary goal is to fine-tune these models using a publicly available dataset containing URLs labeled as either benign or malicious. The different implementations should utilize different techniques, such as standard learning with a cross-entropy loss function and reinforcement learning using the Transformer Reinforcement Learning library. The models should be evaluated using metrics such as accuracy, F1 score, and precision to determine their effectiveness. Additionally, the dataset should be pre-processed using relevant libraries like Pandas, and the results should be visualized with tools such as Matplotlib.

6.1.2. Non-functional requirements. The implementation of the models should be efficient, ensuring that the training and evaluation processes are performed within a practical amount of time while achieving high accuracy and precision. The models should be scalable, capable of handling large datasets in a timely manner. The code should be well-documented and organized in Jupyter notebooks to ensure readability and ease of use. Finally, the implementation should be flexible, allowing for easy experimentation with the different models, parameters and training techniques to optimize performance.

6.2. Design

The technical deliverable is structured into 4 Jupyter notebooks, each independently handling the pre-processing of data and training of different models for URL classification. Namely, the models chosen to perform this task are 1 feed-forward neural network and 3 large language models - BERT, DistilBERT and GPT-2. The main idea behind this choice is to compare the performance on the given classification task between a standard neural network and a more complex LLM, as well a comparison between the different LLMs themselves.

The design of the notebooks' content follows a very similar structure for each of them. First, the same data is used and is suitably pre-processed as per the needs of each model to ensure its format is usable as its input. Next, the data is split into training, validation and test sets and the corresponding model is initialized. This is followed by a training phase adjusted to the respective model in terms of parameters and algorithm used. After training the model for the URL classification task, its performance is evaluated, focusing on metrics like accuracy,

F1 score and precision to determine the effectiveness of its ability to distinguish between benign and malicious URLs. The results are also visualized with tools to generate plots that illustrate the performance metrics to give a clear indication of the model performance.

Each notebook operates as independently, from data pre-processing through to model training, evaluation, and visualization. This independent design allows for flexibility in experimenting with different models, parameters and techniques without the need of having to modify another file to do so.

The project was developed over the span of roughly 3 months. A significant portion of this time was dedicated to creating and refining the code in each notebook. The development process involved iterative improvements based on the evaluation results, including choice of LLM, parameter values, training algorithms and evaluation steps.

6.3. Production

All Jupyter notebook files that the final deliverable consists of utilize the same dataset for training and evaluating the respective model. The dataset in question is a comprehensive collection of 651 190 URLs, openly provided by a public Github repository, hosting a project that is also centered around malicious URL detection [1]. Each URL entry is labeled with various attributes about the structure of the URL and one out of four classification types - *benign, defacement, phishing* or *malware*. Due to the sheer size of the data, in reality only a small but sufficient portion of it is used. Specifically, during the data pre-processing phase in each file, the data is randomly sampled such that only 10 000 arbitrary entries are considered and this sample makes up the data that the model is trained and evaluated on. Moreover, another common pre-processing step is the collective grouping of the 3 non-benign URL classification types (defacement, phishing and malware) into a single label. This is done because for the purpose of this project knowledge of the specific subtype of a malicious URL is not relevant, we only need to know that it is malicious. The use of only 2 labels, *benign* and *malicious* (Fig. 1), also eases the classification task, as this reduces the number of classes, implying a binary classification instead of a more intricate multiclass classification.

The first file of the deliverable (hereafter simply referred to as File 1) implements a basic feed-forward neural network. The notebook begins with code that uses the Pandas library's utilities to read the data from its *csv* file and extract 10 000 random entries from it into a Dataframe for clear visualization and further processing. The next pre-processing steps include removing any potential rows with missing values and setting boolean labels for the entries by appending a new column to the data to serve that purpose, where a value of 1 corresponds to an entry with a classification type of either *defacement, phishing* or *malware* and a value of 0 respectively indicates a *benign* URL (Fig. 1). In the case of this project, the "interesting" URLs are the malicious ones, which is why they are labeled with a positively associated value of 1, despite generally carrying a negative association. Since by doing this new labeling the ***type*** column becomes redundant, it is dropped from the Dataframe. Finally, the features and the labels are extracted into separate corresponding variables and the features are further processed by Sklearn's *StandardScaler* to transform them by performing standardization. As the scaler is unable to do transformations on string data, the ***url*** column is not considered for standardization and is therefore also dropped.

Following the pre-processing steps, the data is ready to be used for training and evaluating the neural network. The next task is to actually build and initialize the model, which is done with the help of Keras module in a separate function. In particular, a basic Sequential structure is used to group and connect 3 Dense layers - input layer, hidden layer and output layer - for the construction the neural network (Fig. 2). The first two layers each have a number of neurons equal to the number of features and the hidden layer uses the ReLU activation function, while the output layer has a single neuron with a sigmoid activation function, suitable for binary classification tasks. Additionally, Dropout layers are added after each of the first two Dense layers to prevent overfitting by randomly deactivating 50% of the neurons during training. L2 regularization is also applied to the Dense layers to further minimize the chances of overfitting by penalizing large weights. The features and labels are then converted into Numpy arrays to ease computations. To ensure that each class is equally represented during training and further make sure that overfitting is less likely to occur, the code makes use of Sklearn's *compute_class_weight* function. This function computes the weights for each class, which are to be used in balancing the classes during the training process. The computed class weights are stored in a dictionary, which maps each class to its corresponding weight. The model is then compiled using the Adam optimizer as the compiler of choice, since it is generally efficient and effective in training neural networks. The loss function used is binary cross-entropy and accuracy is specified as the metric to evaluate the model's performance during training and testing, both of which are standard practices for binary classification tasks.

To train the model and evaluate its performance, 5-fold cross-validation is used. This technique splits the data train, test and validation subsets and 5 folds while ensuring that each fold has a similar distribution of class labels. For each fold, the model is trained on the training subset for up to 10 epochs, with a batch size of 64. These parameter values were deemed to be suitable after extensive tests with different value combinations. The class weights are passed to the fit method to ensure that the model gives appropriate importance to each class during training. To further ensure that overfitting would not occur, an early stopping mechanism is utilized to monitor the accuracy and stop the training process if the accuracy does not improve significantly for 3 consecutive epochs. The model's performance is evaluated on the validation subset after training and the loss and accuracy metrics are recorded for each fold (Fig. 3). Predictions are made

on the test subset and confusion matrices are computed and stored to evaluate the classification performance. In the end, the average accuracy and loss across all folds are computed and printed and a classification report is generated to provide a detailed evaluation of the model's performance on the test set, including precision, recall, and F1-score for each class. Visualizaions of the training history for accuracy and loss as well as the computed confusion matrices for each fold are also displayed by utilizing the Matplotlib and Seaborn modules. The training history plots show the change of accuracy and loss per epoch, while the confusion matrices are suitably annotated to visualize the true positive, true negative, false positive and false negative rates in the evaluation process of classifying a URL as malicious.

The results demonstrate very strong performance across all metrics (Fig. 4). The average accuracy across five folds is 99.12%, with the highest accuracy being 99.60% and the lowest at 98.50%. The average loss is relatively low at 0.026, indicating minimal error, with the highest loss at 0.040 and the lowest at 0.020. The classification report further also suggests near-perfect classification by showing a precision, recall, and F1-score of between 0.98 and 1.00 for both classes. The model's weighted average accuracy is also reported to be 0.99. The training history plots for accuracy and loss per epoch reveal that the model converges very quickly at the second epoch and after this point the improvement starts to become negligible (Fig. 5). The confusion matrices from each fold confirm the very high true positive and true negative rates and close to none false positives and false negatives (Fig. 6). These results collectively indicate that the neural network model is highly reliable for detection of malicious URLs.

The second file of the deliverable, accordingly referred to as File 2, implements a pre-trained large language model and fine-tunes it for the URL classification task, unlike File 1 which used a light implementation of a feed-forward neural network from scratch and was able to be trained on the training data immediately. The LLM in question is BERT and since it has already been trained on a considerable amount of various data, the expectations are that its performance would be noticeably reduced compared to the previous model.

The file first loads and reads the dataset in exactly the same way as before. Then once again the binary label column is created and appended to the Dataframe, while the redundant ***type*** column is dropped. These initial steps are followed by further pre-processing of data specific to the BERT model (Fig. 7). First, the categorical type labels are converted into numerical codes suitable for the model. The data is then split into training, testing and validation sets using Sklearn's *train_test_split* method to prepare it for training and evaluation later. Next, since the BERT model is expected to perform classification based on the textual data in the dataset rather than the numerical features, that data must be split into tokens so that it is converted into a format that can be processed by the model. For clarity and reusability the tokenization process is implemented in a function that uses a selected tokenizer to generate input IDs and attention masks from the data in the ***url*** column for each URL, which are necessary for BERT to understand the input sequences. To ensure a better performance of the model, a DataLoader is utilized to implement efficient loading and shuffling of the input data in batches, which is once again wrapped into its own function for the same reasons. This is followed by calling the two functions for both the training and validation data. The data is tokenized using the default BERT tokenizer and the data loaders are initialized for passing the input data to the model into batches of 8. This number was chosen after several tests were conducted with different batch sizes (4, 8, 16, 32 and 64) and it was evident that it provided the fastest performance for the model without it showing signs of underfitting or overfitting.

A BERT model for sequence classification, suitable for the binary classification task at hand, is then initialized with default pre-trained weights and the number of output labels is set based on the unique labels present in the dataset, which in this case is 2. The model is then moved to the appropriate device (GPU if available, otherwise CPU) for training. In practice, the device used throughout the process was always the GPU, as the code was run on a computer with NVIDIA GeForce GTX 1650 Ti, which is a CUDA-compatible GPU, meaning that it could make use of NVIDIA's CUDA platform to train the model using the GPU's processing power and achieve faster performance through more optimized computations. Nevertheless, the CPU option remains for the cases where the code might be executed on a machine without a CUDA-supported GPU or if a failure related to the GPU occurs. Continuing the set up before the training and evaluation stages, an AdamW optimizer is initialized, setting the model's learning rate at 0.000001. The AdamW optimizer was picked since it is a common choice for fine-tuning BERT models, as it mitigates overfitting by combining the benefits of the Adam optimizer with weight decay regularization. Moreover, the final selection of the learning rate value was once again determined through a process of trial and error to find the value which minimizes the possibility of under- and overfitting the model while at the same time achieving the best possible metric results. Finally, to handle class imbalance in the data, class weights are once again computed using Sklearn's *compute_class_weight* function. These weights are then converted to a tensor and moved to the device to be used during the training process.

The training loop is set up to train the model and evaluate it over multiple epochs. Several tests with different values for the maximum number of epochs confirm that the overall performance does not improve significantly past the fifth epoch. Therefore the final number of epochs was picked to be 5. During each epoch, the model is set to training mode, and the training data is processed in batches. For each batch, input IDs, attention masks, and labels are moved to the device. Gradients are reset, model outputs are computed, and the loss is calculated using the cross-entropy loss function. The gradients are backpropagated, and the optimizer updates the model parameters. During the validation phase, the model is set to evaluation mode, the validation data is processed in

batches without updating the model parameters and predictions are generated on it. To monitor the performance, the validation loss is computed and further evaluation metrics including accuracy, precision, recall, and F1-score are calculated. A confusion matrix is also generated and plotted to visualize the performance of the model on the validation set. A simple early stopping mechanism has also been implemented that prevents the execution of further epochs by halting the loop as soon as the computed validation accuracy starts decreasing, which is a common sign of overfitting. Picking the accuracy as the metric to monitor for this is a safe choice because the class weights applied to the data ensure that there would be no accuracy bias caused by a potential class imbalance.

The evaluation results show that the model demonstrates strong performance with high accuracy, precision, recall, and F1-scores across all epochs. Similarly to the feed-forward neural network from File 1, the BERT model appears to learn exceptionally quickly, indicated by the high accuracy (93.45%), precision (93.80%), recall (93.45%) and F1-score (93.51%) right after the first epoch (Fig. 8). At the next epoch there is a small but noticeable increase in all of those metrics by about 2% each as well as a notable decrease in the validation loss from 0.2025 to 0.1304, indicating a noteworthy improvement in performance. However, from epoch 2 through epoch 5 the total increase in accuracy (95.85% to 97.60%) and decrease in validation loss (0.1304) are around the same as the ones between epoch 1 and epoch 2, implying that the model is not able to improve its performance significantly in further epochs because it has already learned to classify the data effectively during the first two (Fig. 9). This is also demonstrated by the confusion matrices, which show a consistent decrease in both false positives and false negatives across epochs, albeit at a visibly lower rate in the last three epochs. These results give confirmation that the BERT model is very much suitable for URL classification, as it makes reliable predictions once it is fine-tuned for this task.

The third file, referred to as File 3, implements another pre-trained LLM fine-tuned for URL classification. This time the particular model chosen is DistilBERT [8], which, as the name suggests, is a distilled form of BERT. This means that in essence it is a BERT-based model developed after the application of a distillation process, which has reduced its size (40% smaller) and number of parameters (40% less parameters) while retaining almost exactly the same performance (roughly 97% similarity) and simultaneously achieving greater speed (60% faster) during the training phase. As such, the steps for data preparation, training and evaluation for this model are exactly the same as for the BERT implementation in File 2. The parameter values for the learning rate, batch size and the maximum epochs also remain the same for the purpose of comparing the the two models under the same conditions, as DistilBERT is expected to exhibit a slightly lower but very close performance to that of BERT. These expectations are confirmed by the results, which once again show very high metric values (94.05% accuracy, 94.21% precision, 94.05% recall, 94.09% F1-score) after the initial epoch (Fig. 10) and a slow but steady increase of the accuracy (94.05% to 97.40%) and decrease of the validation loss (0.1892 to 0.0706) throughout the epochs (Fig. 11). This is further reflected in the confusion matrices, which once more show a notable decrease in the false positives and false negatives for each successive epoch. These results indicate that DistilBERT, while being smaller and using less computational power, still maintains a high level of performance comparable to that of BERT, making it a highly effective model for the URL classification task.

The fourth and final file, respectively referred to as File 4, is supposed to contain an implementation of a LLM with a transformer-based architecture (like the previously seen BERT and DistilBERT) trained for the classification task using the Transformer Reinforcement Learning (TRL) library provided by HuggingFace [9]. The idea behind utilizing TRL was to explore its potential when integrated with a LLM for classification. The attempt at implementing this approach was mostly experimental, due to the lack of clear documentation and usage examples for the module. As such, the implementation produced many exceptions throughout development and was unfortunately not successful, leaving only a partially implemented code in the end. Since File 4 is an incomplete implementation, it will not be described in so much detail as the previous files and only the most important sections will be mentioned.

The first few steps share a lot of similarities to the File 2 and File 3. The file begins by reading and processing the data and its labels in the same way as before. This time the choice of LLM to fine-tune for the classification task was GPT-2, simply because the documentation of TRL had included examples of its integration with GPT-2 in particular. To prepare the data for correct processing by the model, the GPT-2 tokenizer from HuggingFace is initialized to tokenize the URLs with the same process like before, after which the tokenized data is loaded into DataLoader objects to make use of batch processing during training and evaluation once again. This is followed by the initialization of a GPT-2 model with a value head to integrate reinforcement learning capabilities and the model is configured and moved to the suitable device, either GPU or CPU.

The next step sets up the configuration for the reinforcement learning approach. For that it requires a PPO (Proximal Policy Optimization) trainer and a reward model (Fig. 12). PPO is a reinforcement learning algorithm that balances the trade-off between exploration and exploitation by adjusting the policy gradually within a defined threshold. The PPO configuration for this implementation includes setting various hyperparameters, such as learning rate, batch size and gradient accumulation steps, all of which were not extensively tested with values due to the lack of completeness of the implementation. The PPO trainer is initialized with the model, tokenizer, dataset, and PPO configuration. This setup allows the model to be trained using the PPO algorithm, where the model's policy is iteratively updated based on the rewards received from the reward model. As the name would suggest, a reward model is used to provide feedback to the agent based on its actions (in

this case, the responses generated during training) and helps in guiding the training process by assigning scores to the model's outputs. The reward model used in this implementation is DistilBERT, which was selected because of its efficiency, faster inference times, and reduced computational demands, while retaining high performance in text classification tasks, as already demonstrated earlier by File 3. This balance makes it ideal for providing accurate and sufficiently impactful rewards in the reinforcement learning process.

Finally, the training and evaluation is supposed to be performed over multiple epochs following the same overall steps as with previous models: training the model on the training data, generating responses, while also computing rewards and performing PPO steps; evaluating the model on the validation data, computing metrics such as loss, accuracy, precision, recall and F1-score; and in the end plotting confusion matrices to visualize the model's performance. Unfortunately several exceptions were encountered in the code for this loop, most of which were related to either the batch size of the input data for the PPO trainer or its format and shape, leading to the conclusion that the trainer does not process the data properly. Since there was not enough knowledge about the correct approach to resolve this issue and even less time to achieve it, there was no other choice but to leave the implementation incomplete and finalize the deliverable in the current state.

6.4. Assessment

The initial goal of the technical deliverable was to implement and evaluate at least two large language models tailored for URL classification, using different techniques such as standard learning with a cross-entropy loss function and reinforcement learning using the Transformer Reinforcement Learning library. The developed programs and their results mostly accomplish this goal, apart from the incomplete outcomes in regard to the implementation of an LLM with a reinforcement learning approach. Because of this, the deliverable manages to provide better insight about the task performances of a feed-forward neural network and two closely related transformer-based LLMs, namely BERT and DistilBERT. It also successfully makes use of the information provided in the dataset to fine-tune the LLMs, thereby providing a conclusive answer to the project's objective of evaluating the effectiveness of LLMs in URL classification. Each model was tested and evaluated using standard metrics like accuracy, F1 score, and precision. The results indicate that all of them perform exceptionally well.

A notable finding is that, while being smaller and lighter than BERT, DistilBERT still managed to perform just as effectively as its more computationally able counterpart. At the same time, DistilBERT's inference speed also serves as a crucial aspect for the proper evaluation of the model, as the time spent per epoch during training was roughly twice as less than that of the BERT model (nearly 6 minutes, compared to BERT's 12 minutes), practically achieving the same goal under the same conditions much faster. Therefore, it is safe to say that, in the context of malicious URL detection, DistilBERT would be just as effective of a choice as BERT, potentially even more appropriate due to its similar performance and much better training speed.

One major area that showed potential for improvement was the experimental implementation using TRL with GPT-2 in File 4. Although the initial setup and configuration were successful, the implementation encountered several exceptions related to data processing and batch sizes. These issues prevented the completion of the reinforcement learning training loop. Had more time and resources been available, further exploration of potential solutions and of the TRL documentation could have been done to fully realize its potential.

As a whole, for the most part the technical deliverable meets the specified functional and non-functional requirements and produces the expected results. The implementations clearly demonstrate the feasibility and effectiveness of using LLMs for URL classification with high performance metrics. Despite the incomplete implementation of the reinforcement learning approach, the overall success in achieving the main goals of the deliverable suffices to deem it an overall success, albeit with clear potential for further improvement.

Acknowledgment

The authors would like to thank the BiCS management and education team for the amazing work done.

7. Conclusion

The research and analysis presented in this report have demonstrated the potential and practicality of using large language models to detect malicious URLs. The scientific exploration covered essential concepts such as fine-tuning pre-trained LLMs, data pre-processing and model performance evaluation, all of which are crucial for applying LLMs to cybersecurity tasks. The deliverable also gave a discussion about past use cases, thus providing a comprehensive context for the application of LLMs in identifying and mitigating cyber threats.

The technical deliverable further supported the scientific findings by implementing and evaluating several LLMs tailored for URL classification. Through the detailed descriptions of model training, fine-tuning and evaluation processes, the deliverable showcased the practical steps required to adapt LLMs for detecting malicious URLs. Apart from the challenges encountered with the reinforcement learning approach, the models demonstrated very high performance, confirming the feasibility and effectiveness of LLMs in this domain.

In conclusion, the project has successfully shown the ways in which LLMs can be effectively utilized for detecting malicious URLs. The research has clearly demonstrated that integrating theoretical knowledge with practical implementations, and with proper fine-tuning and optimization, can make LLMs powerful tools in cybersecurity for mitigating the impact of evolving threats.

References

[1] https://github.com/Priyanshu9898/End-to-End-Malicious-URL-Detection/tree/main

[2] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[3] Raja, A. Saleem, R. Vinodini, and A. Kavitha. "Lexical features based malicious URL detection using machine learning techniques." Materials Today: Proceedings 47 (2021): 163-166.

[4] Abdi, Farhan Douksieh, and Lian Wenjuan. "Malicious URL detection using convolutional neural network." Journal International Journal of Computer Science, Engineering and Information Technology 7.6 (2017): 1-8.

[5] Verma, Rakesh, and Avisha Das. "What's in a url: Fast feature extraction and malicious url detection." Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics. 2017.

[6] Sahoo, Doyen, Chenghao Liu, and Steven CH Hoi. "Malicious URL detection using machine learning: A survey." arXiv preprint arXiv:1701.07179 (2017).

[7] Le, Hung, et al. "URLNet: Learning a URL representation with deep learning for malicious URL detection." arXiv preprint arXiv:1802.03162 (2018).

[8] https://huggingface.co/docs/transformers/model_doc/distilbert

[9] https://huggingface.co/docs/trl/index

8. Appendix

```
data_cleaned['type_binary'].value_counts()
```

```
type_binary
0    6521
1    3479
Name: count, dtype: int64
```

Fig. 1: Distribution of benign (labeled as 0) and malicious (labeled as 1) URLs in the input data

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

def create_model(input_shape):
    model = Sequential([
        Dense(input_shape, activation=None, input_shape=(input_shape,), kernel_regularizer=None),
        Dropout(0.5),
        Dense(input_shape, activation='relu', kernel_regularizer=None),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Fig. 2: Feed-forward neural network design

```
Fold 1:
Epoch 1/10
125/125 [==============================] - 1s 3ms/step - loss: 0.4225 - accuracy: 0.8227 - val_loss: 0.1237 - val_accuracy: 0.9785
Epoch 2/10
125/125 [==============================] - 0s 2ms/step - loss: 0.1938 - accuracy: 0.9415 - val_loss: 0.0493 - val_accuracy: 0.9885
Epoch 3/10
125/125 [==============================] - 0s 2ms/step - loss: 0.1162 - accuracy: 0.9670 - val_loss: 0.0272 - val_accuracy: 0.9935
Epoch 4/10
125/125 [==============================] - 0s 2ms/step - loss: 0.0740 - accuracy: 0.9775 - val_loss: 0.0197 - val_accuracy: 0.9960
63/63 [==============================] - 0s 1ms/step
63/63 [==============================] - 0s 1ms/step - loss: 0.0197 - accuracy: 0.9960
Score: loss 0.01974928192794323; accuracy 0.9959999918937683

Fold 2:
Epoch 1/10
125/125 [==============================] - 1s 3ms/step - loss: 0.5849 - accuracy: 0.7514 - val_loss: 0.2240 - val_accuracy: 0.9580
Epoch 2/10
125/125 [==============================] - 0s 2ms/step - loss: 0.2306 - accuracy: 0.9250 - val_loss: 0.0884 - val_accuracy: 0.9765
Epoch 3/10
125/125 [==============================] - 0s 2ms/step - loss: 0.1230 - accuracy: 0.9631 - val_loss: 0.0519 - val_accuracy: 0.9825
Epoch 4/10
125/125 [==============================] - 0s 2ms/step - loss: 0.0881 - accuracy: 0.9728 - val_loss: 0.0404 - val_accuracy: 0.9850
63/63 [==============================] - 0s 1ms/step
63/63 [==============================] - 0s 1ms/step - loss: 0.0404 - accuracy: 0.9850
Score: loss 0.040367502719163895; accuracy 0.9850000143051147
...
63/63 [==============================] - 0s 1ms/step
63/63 [==============================] - 0s 1ms/step - loss: 0.0277 - accuracy: 0.9910
Score: loss 0.027724767103791237; accuracy 0.9909999966621399
```

Fig. 3: Monitoring the training process of the feed-forward neural network

```
print('Average score:')
print(f'Accuracy: {np.mean(accuracy)} (max = {np.max(accuracy)}, min = {np.min(accuracy)})')
print(f'Loss: {np.mean(loss)} (max = {np.max(loss)}, min = {np.min(loss)})')
```

✓ 0.0s

```
Average score:
Accuracy: 0.9911999940872193 (max = 0.9959999918937683, min = 0.9850000143051147)
Loss: 0.026418504118919373 (max = 0.040367502719163895, min = 0.0196960214525462)
```

```
from sklearn.metrics import classification_report

print('Classification report:')
print(classification_report(labels[test], predicted_labels))
```

✓ 0.0s

```
Classification report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      1326
           1       0.98      1.00      0.99       674

    accuracy                           0.99      2000
   macro avg       0.99      0.99      0.99      2000
weighted avg       0.99      0.99      0.99      2000
```

Fig. 4: Metrics results for the feed-forward neural network

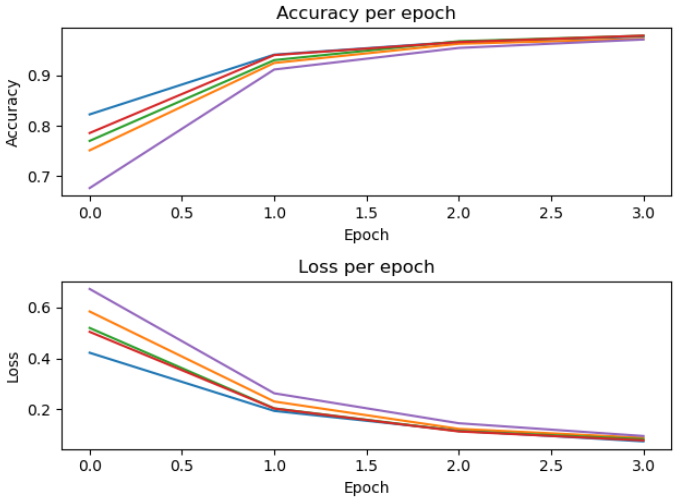

Fig. 5: Accuracy and loss per epoch plots for the first 3 epochs of the feed-forward neural network

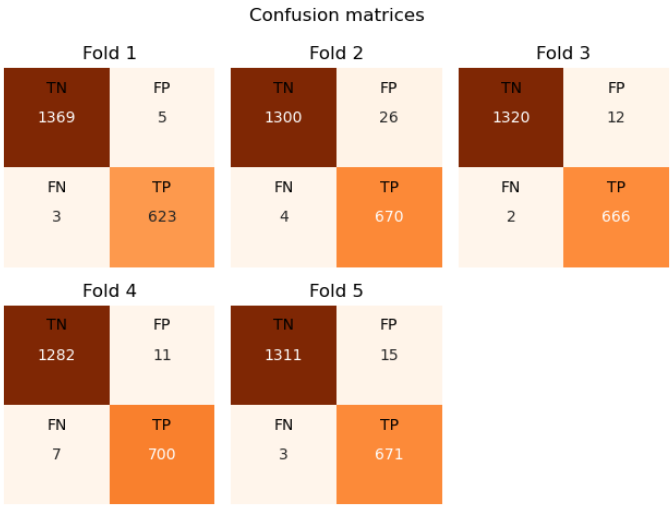

Fig. 6: Confusion matrices for the feed-forward neural network

```
def prepare_data(data, tokenizer, max_len=256):
    tokens = tokenizer.batch_encode_plus(
        data['url'].tolist(),
        max_length=max_len,
        padding='max_length',
        truncation=True,
        return_tensors='pt'
    )
    return {
        'input_ids': tokens['input_ids'],
        'attention_mask': tokens['attention_mask'],
        'labels': torch.tensor(data['label'].values, dtype=torch.long)
    }


def create_data_loader(dataset, batch_size):
    return DataLoader(
        [{key: value[i] for key, value in dataset.items()} for i in range(len(dataset['labels']))],
        batch_size=batch_size,
        shuffle=True
    )
```

Fig. 7: Functions for preparing the data to be processed by the BERT model

```
Starting epoch 1/5
100%|██████████| 1000/1000 [11:39<00:00,  1.43it/s]
Epoch 1, Validation Loss: 0.2025
Accuracy: 0.9345, Precision: 0.9380, Recall: 0.9345, F1 Score: 0.9351
```

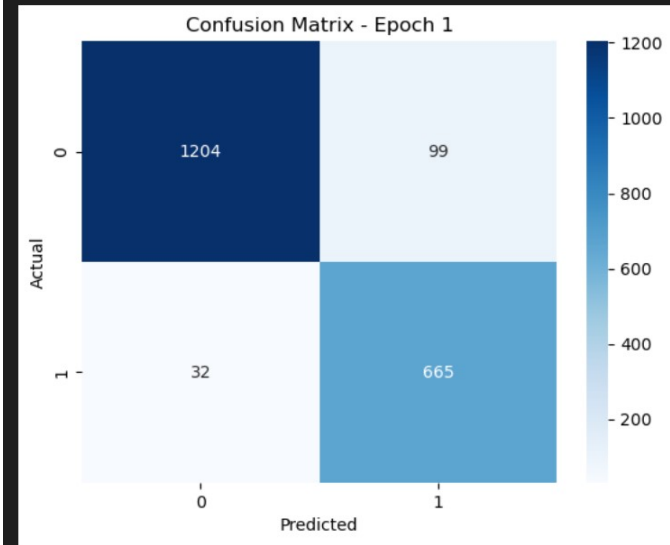

Fig. 8: Evaluation results for BERT after one epoch

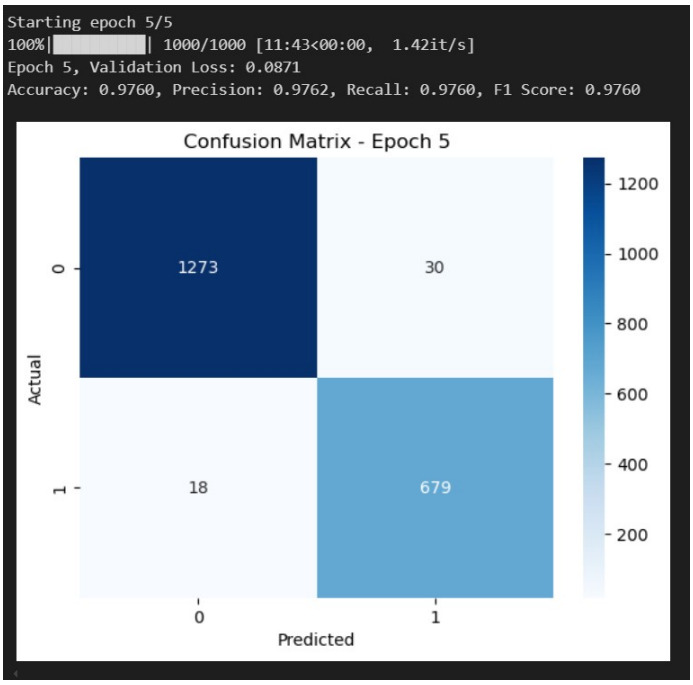

Fig. 9: Evaluation results for BERT after five epochs

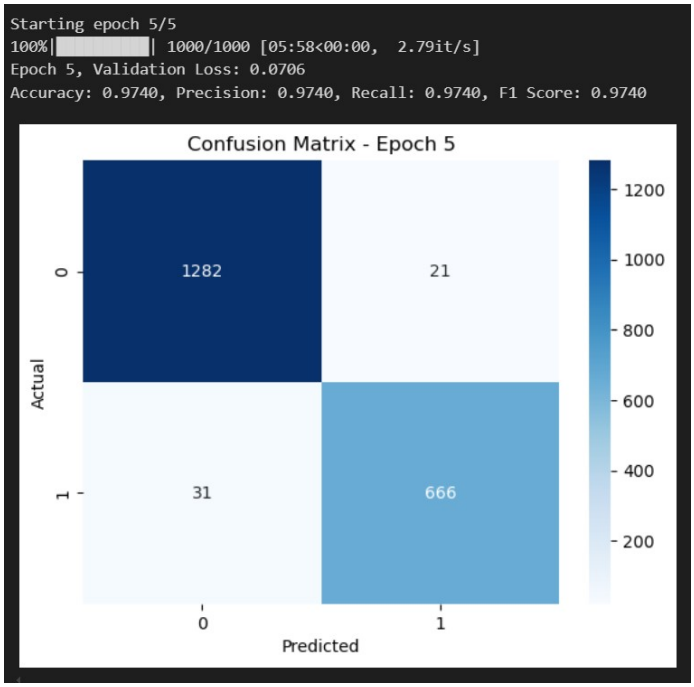

Fig. 11: Evaluation results for DistilBERT after five epochs

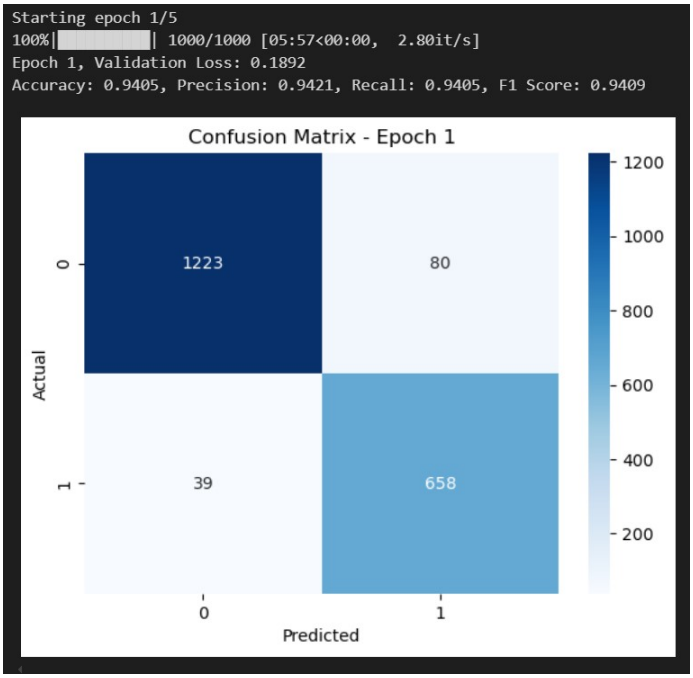

Fig. 10: Evaluation results for DistilBERT after one epoch

```python
# PPO config
ppo_config = PPOConfig(
    model_name='gpt2',
    learning_rate=0.000001,
    batch_size=32,
    forward_batch_size=8,
    mini_batch_size=8,
    gradient_accumulation_steps=4
)

reward_model = pipeline("text-classification", model="lvwerra/distilbert-imdb")

ppo_trainer = PPOTrainer(
    model=model,
    config=ppo_config,
    tokenizer=tokenizer,
    dataset=train_dataset
)
```

Fig. 12: PPO configuration and reward model setup