A Transformer-based Approach for Vulnerability Detection

José María Salvador

Received June 06, 2023
Accepted August 01, 2023
Electronic access September 30, 2023

Over the past years, cybersecurity has become an increasingly important aspect of protecting sensitive information and systems from malicious attacks. As mobile and web applications become more relevant within our everyday life, it is vital to continuously develop techniques that are capable of protecting these systems. Vulnerability analysis is a critical aspect of cybersecurity, as it helps identify and mitigate potential security risks within software systems. While most recent techniques rely on Deep Learning, few are able to take advantage of the recently developed transformer models and their self-attention mechanism. In this paper, a comprehensive account is presented, outlining everything from data pre-processing to hyperparameter customization, for training a transformer neural network in the task of vulnerability detection, achieving an accuracy of over 90% in real-world data. This will allow developers to identify and address vulnerabilities early in the development life-cycle, saving time and effort by avoiding the need for extensive rework, and strengthening the overall resilience of software systems.

Introduction

Web and mobile applications have become a fundamental part of modern life, allowing their users to easily access a wide range of services, such as information or entertainment. Although this poses a vast number of benefits, it also comes with a lot of risk; more specifically, cybersecurity threats. A lot of these applications can unintentionally open a pathway for security breaches, mainly due to internal flaws such as improper data handling, coding errors, or weak authentication mechanisms. This prompts us to take proactive measures that may help mitigate the said risks, such as vulnerability analysis, a process that focuses on identifying and assessing those internal weaknesses.

Traditionally, vulnerability analysis techniques have relied on rule-based (where a set of predefined rules are used to find the vulnerabilities) or signature-based (where characteristics, or "signatures" related to a specific threat are identified) methods[1]. These can have limited effectiveness and flexibility, stemming from dependencies on known signatures, a lack of context awareness, and a considerable amount of false positives and false negatives. In order to overcome these limitations, researchers have explored the use of deep learning techniques, such as convolutional neural networks (CNNs) and long short-term memory networks (LSTMs)[2]. While these techniques have certainly brought major improvement within the field of vulnerability analysis, there are still certain features that could be enhanced with the implementation of attention, as it is done by the transformer model.

An attention model is basically an input processing mechanism, which can be very useful when undertaking complicated tasks, as it is able to divide those tasks into smaller assignments that it can sequentially complete. It works by essentially enhancing certain parts of the input data, while diminishing others. This comes with many benefits, including the enhanced capability of capturing the global context of the input, given that these models excel at capturing long-range dependencies in input data. Fields such as Natural Language Processing (NLP) have greatly benefited from these models[3], showing that they are well-suited for processing large amounts of text data, such as the one that can be found in code.

Our intuition is that appropriate implementation of self-attention mechanisms could improve the accuracy, efficiency, and interpretability of vulnerability analysis techniques. This project aims to demonstrate the effectiveness of these architectures and provide insights into their potential impact on the field by adapting the RoBERTa pre-trained transformer model for vulnerability detection. The said model is both trained and tested on real-world data, enabling it to capture its intricacies and nuances, leading to predictions that are more accurate and relevant.

Background

Code Tokenization

Code tokenization is a process that involves breaking down code into individual tokens, where each token represents a certain element of the code. The dividers usually consist of keywords, identifiers, literals, and operators, as can be observed in Figure 3. In the said case, identifiers such as 'int' and 'char', as well as operators such as '*' and '=' are left the same way after the tokenization. On the other hand, functions such as 'curl mvsprintf' and variables such as 'retcode' are simplified

© The National High School Journal of Science 2023

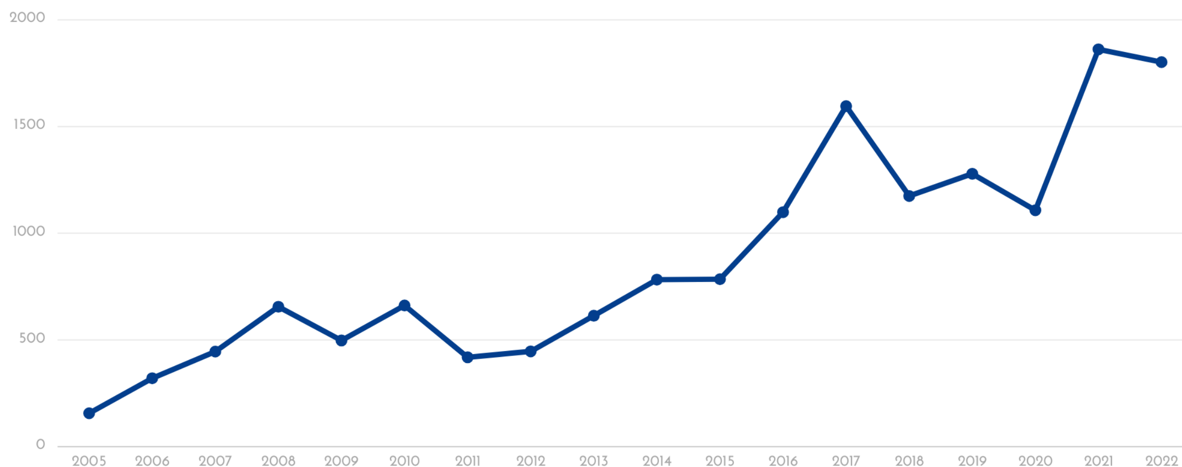

Fig. 1 Annual number of data compromises in the United States from 2005 to 2022.[4]

to tokens such as 'FUNC1' and 'VAR5', respectively. This makes it much easier for the system to analyze and process the code.

Software Vulnerability Analysis

Software vulnerability analysis is the process of systematically searching for weaknesses in an information system. The purpose of this assessment is to reduce the likelihood of potential cyberattacks. This usually involves assigning severity levels to as many security flaws as possible, usually based on glossaries such as CVE or CVSS, with both manual and automated techniques that may vary in levels of rigor and efficiency[5].

Static Analysis

Static analysis is a software testing technique that is carried out without actually having to execute the code. It analyzes components such as program structure, control flow, and data flow to identify potential vulnerabilities. While there have been various implementations of static analysis for vulnerability detection[6], they usually come with low accuracy, mainly manifested within their high numbers of false positives. Despite this, there have been successful attempts to significantly reduce the number of false positives produced[7].

Dynamic Analysis

Dynamic analysis relates to a set of techniques used to test and evaluate a program's behavior during runtime. Some of these techniques include fuzzing[8] and penetration testing[9]. This type of analysis is usually able to uncover bugs that would be too complicated for static analysis to reveal, though it still suffers from a high false-positive rate.

ML-Based Approaches

Machine learning has emerged with multiple promising techniques for vulnerability analysis. These techniques are able to analyze large volumes of data to identify patterns and abnormalities, as well as continuously learn and improve from new input. Many comparative research studies have been conducted[10], showing a wide range of potential methods for vulnerability analysis.

Transformer Model

The transformer neural network is a machine learning model used mainly for natural language processing tasks, such as text summarization, translation, and classification. The transformer model was initially proposed by Vaswani et al.[11], and our model uses that same structure, adapted for vulnerability classification.

One of the transformer's main features is that it can process sequences of input data in parallel, rather than sequentially, as did previous models. This is achieved in great part because

 © The National High School Journal of Science 2023

```
1  int curl_mvsprintf ( char * buffer , const char * format , va_list ap_save ) {
2  int retcode ;
3  retcode = dprintf_formatf ( & buffer , storebuffer , format , ap_save ) ;
4  * buffer = 0 ;
5  return retcode ;
6  }
```

Fig. 2 Example of a code snippet

```
['int', 'FUNC1(', '(', 'char', '*', 'VAR1', ',', 'const', 'char', '*', 'VAR2', ',', '
VAR3', 'VAR4', ')', '{', 'int', 'VAR5', ';', 'VAR5', '=', 'FUNC2(', '(', '&', 'VAR1',
 ',', 'VAR6', ',', 'VAR2', ',', 'VAR4', ')', ';', '*', 'VAR1', '=', 'NUMBER', ';', 'r
eturn', 'VAR5', ';', '}']
```

Fig. 3 Code snippet from Figure 2 after being tokenized and passed onto a list.

of its pioneering self-attention mechanism, which allows it to selectively attend to different input data simultaneously.

A vital part of this self-attention mechanism is its multi-head attention layer. This consists of dividing our input into multiple heads, where each head has its own set of weight parameters. This allows us to run through the mechanism several times in parallel, rather than progressively going through all of the input.

Additionally, within every head, each word in a sequence attends to all other words in the same sequence to determine their relative importance for the task at hand. The mechanism can be represented by the following equation:

$$\text{Self-Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where Q, K, and V are the matrices for queries, keys, and values, respectively, with dimensions $n \times d_k$, $m \times d_k$, and $m \times d_v$. n is the length of the input sequence, m is the maximum length of all sequences in the dataset, d_k and d_v are the dimensions of the keys and values, respectively. The dot product of Q and K is divided by $\sqrt{d_k}$ to reduce their variance and then passed through a softmax function to obtain the scaled attention weights. Finally, the said attention weights are used to determine the values matrix V, producing the output of the self-attention mechanism.

In addition to this, the Transformer is divided into two parts; the encoder and decoder. The encoder is mainly focused on processing the input sequence and capturing the contextual information of each token, whilst the decoder is responsible for generating a target sequence based on the contextualized representations produced by the encoder. While both components consist of stacks of identical layers, the decoder differs in the fact that it has an additional cross-attention mechanism that allows it to attend to the encoder's output.

Related Attention-Based Models

There have been multiple different approaches for vulnerability detection using Deep Learning techniques, more specifically, using models that implement attention. Some of these methods are based on code representation graphs, leveraging Graph Neural Networks (GNNs) to better capture the semantics and informational aspects embedded within source code. This particular technique allowed for a high accuracy and F1 score when testing on the SARD-NVD database[12]. Others have presented modified versions of pre-trained models, tweaking certain aspects to adapt them for vulnerability detection, similar to how we did. An example of this is VulBERTa[13], which utilizes a low-complexity model with a custom tokenization pipeline. The said model achieved a high precision score on the Vuldeepecker dataset, though presented an accuracy of 84.48% when testing with the ReVeal dataset. Certain studies have even combined different architectures with self-attention mechanisms to improve their model's capabilities, such as Chen et al.'s[14] implementation of a self-attention mechanism within Temporal Convolutional Networks (TCNs) for their vulnerability detector, AIdetectorX.

While all of these models have excelled in their respective environments, one main area of improvement is their accuracy with real-world datasets. This is arguably one of the most important points that the model needs to cover so that it can actually be implemented for applied cybersecurity purposes. Most of the aforementioned techniques either saw a considerable drop in accuracy when testing their model on real-world data, or did not test it at all with a real dataset within their paper. While some reached relatively high accuracies of around 85%, we believed there was still a lot of room for improvement

Methods

Environment Configuration

For the implementation of this model, we decided to use Google Colab Pro[15]. This gave us access to plenty of cloud-based computational resources. The GPUs were especially useful, as the transformer model required them in order to run the CUDA environment. The ability to leverage multiple GPUs or TPUs also made it easier to run our model, as well as more scalable. A slight disadvantage of using Google Colab, though, was that it would have an idle timeout time of around 90 minutes, so if we stopped interacting with the model it would stop running the given cell.

Dataset and pre-processing

The model that we trained required a large database of code snippets that were classified as either vulnerable or non-vulnerable. These datasets can be labeled as synthetic, semi-synthetic, or real data. Synthetic datasets are those that artificially generate annotations and code snippets. Rather similarly, within semi-synthetic data, either the code or the annotations are artificially generated. The main issue with these types of datasets is that they can often lead to biased results, as they are not able to capture the full range of variability in real-world data.

As a result, we trained our model with real data, where both the code and the annotations are extracted from real-world sources. More specifically, we used the ”ReVeal” data collection[16], a curated dataset obtained by tracking past vulnerabilities from Linux Debian Kernel and Chromium. This data collection contains a total of 18,169 total code samples, with slightly over 9% being labeled as vulnerable (as can be visualized in Figure 5). This enhanced the realism, applicability, and effectiveness of our model, given that its challenges and characteristics are aligned with those of real-world scenarios.

To prepare the data for our model, we first downloaded both the vulnerable and non-vulnerable JSON files and parsed them so that we only kept the code snippets on a list object. Additionally, every code snippet within the list had to be accompanied by either a ”0” or a ”1” to indicate whether the code was non-vulnerable or vulnerable, respectively. Afterward, the list was shuffled with Python's ”random.shuffle” function to reduce any possible bias and prevent order-related patterns to form. With this newly shuffled list, we applied scikit-learn's ”train test split” function to split our data into a training and a testing list, with an 80:20 ratio. Ultimately, each list was individually exported to its own CSV file.

Challenges

For the model's implementation, we attempted multiple techniques and had to work through some issues. Initially, we looked through different ways of tokenizing our code, with varying degrees of success. The first tokenizer we tried was the ”sctokenizer” function, which can be installed as a Python package, but the tokenization was not as concise as we would've wanted it to be. We needed it to be more concise because long tokens can negatively impact the performance of our model. For starters, it increases memory usage and processing times, given that the model is forced to analyze larger strings of data. Additionally, and more importantly, it can lead to a loss of context. This happens because transformer models have a maximum sequence length due to memory constraints, and longer tokens may exceed this limit, necessitating truncation or splitting of the text. This can result in the loss of important information or a disruption in the coherence of the input, impacting the efficiency of the model.

While the function was able to accurately separate the elements of the code into individual tokens, the output list was too long for the model to use, as is demonstrated in Figure 6. We even attempted to cut out certain parts of the output, leaving the tokens only with their value and type indicators: (char-KEYWORD) Despite this, we recognized that the divisions were still not general enough for the classifier to accurately take in and decided to move on with another method, as is described in the following section. This allowed us to forego the manual process of code tokenization, as the library that we utilize automates that process.

Model Architecture

The RoBERTa model implements state-of-the-art transformer architecture, incorporating several improvements and modifications from its predecessor, BERT. With its self-attention mechanism, bidirectional processing, and extensive pre-training, the model has an enhanced capability for capturing long-range dependencies, something particularly effective for vulnerability detection. Additionally, the size and diversity of the model's training corpus contribute to its ability to learn robust representations of language, which can then be passed on to certain features required for this type of context.

Implementation

We followed Hamdaoui's[17] implementation of the transformer model, adapting it for vulnerability classification as can be seen in Figure 7. This uses the ”ClassificationModel” from the Simple Transformers library. To use the said library, we provided the pre-processed labeled data as a CSV file with two columns; one for code and one for the vulnerability label. Additionally, the process was divided into a training and

© The National High School Journal of Science 2023

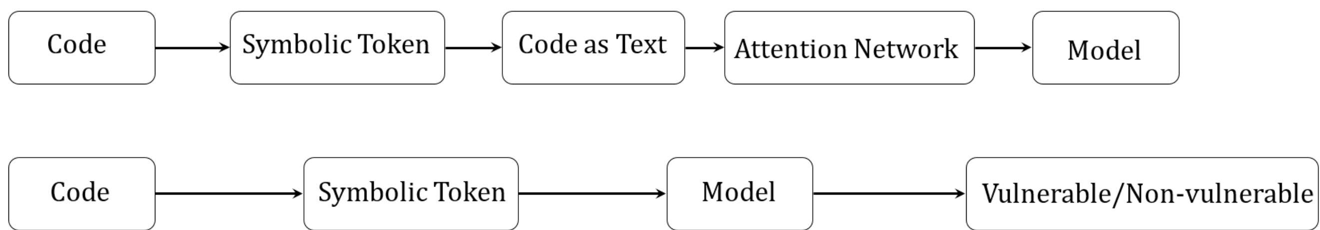

Fig. 4 Block diagrams representing the process of training (top) and testing (bottom)

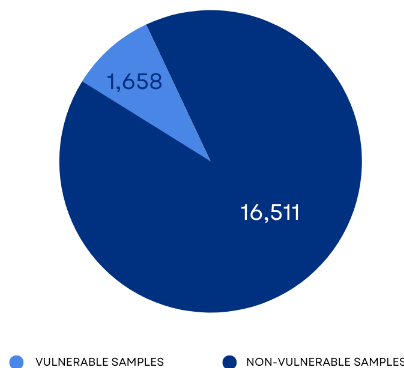

Fig. 5 Visualization of vulnerable against non-vulnerable code samples within the ReVeal dataset

a testing loop, as is demonstrated in Figure 4.

Stratified K-Fold

For the training loop, we split the data using a scikit learn's "Stratified k-fold cross validation" method. With this function, the dataset is randomly divided into "k" splits, or folds, while maintaining the proportions of each target class. This is particularly useful in our case, as the dataset we are using is imbalanced, with a much more significant portion of the data being non-vulnerable code. The model is trained "k" times, with each iteration having a fold being used as a validation set, with the remaining "k-1" folds being used for training. For our case, we decided to define "k=5", meaning we had to train through five splits. Additionally, we defined the "shuffle" parameter as "true", so that the samples from each class are shuffled before splitting them. Finally, we set the "random state" parameter as "1997" for reproducible outcomes.

RoBERTa Model

When initializing the "ClassificationModel", we had to specify the model that we were using. In this case, we used the RoBERTa model (which stands for Robustly Optimized BERT Pretraining Approach), a large neural language processing model introduced by Liu et al. in 2019[18]. It builds upon the BERT (Bidirectional Encoder Representations from Transformers) architecture[19], further optimizing its training methodology and structure. This model has achieved stateofthe-art performances on various NLP tasks such as text classification[20], and entity recognition[21], which is why we decided it would be perfect for the implementation of our model.

Customization and Hyperparameters

The Simple Transformers library handled the tokenization of the code and gave many options for the customization of the model's hyperparameters, as demonstrated by Table 1. For starters, we decided to implement a relatively small train batch size of sixteen for better convergence and generalization per-

```
char * curl_mvaprintf ( const char * format , va_list ap_save )
```

Fig. 6 Example of a code snippet (top) and the output after running the said code through the ”sctokenizer” function (bottom)

formance. Additionally, we set ”overwrite output dir” to ”true” in order to overwrite existing saved models in the same directory, as well as ”reprocess input data” so that input data is reprocessed even if a cached file exists. We also set the number of training epochs to four and the ”max sequence length” to 128, limiting the maximum sequence length that the model will support. Finally, we set a zero weight decay, allowing for faster training and easier interpretation of the model’s weight, as well as a learning rate of 2×10^{-5}.

Hyperparameter	Value
Batch size	16
Overwrite output directory	True
Reprocess input data	True
Training epochs	4
Max sequence length	128
Zero-weight decay	True
Learning rate	2×10^{-5}

Table 1 Hyperparameters used along with their respective values

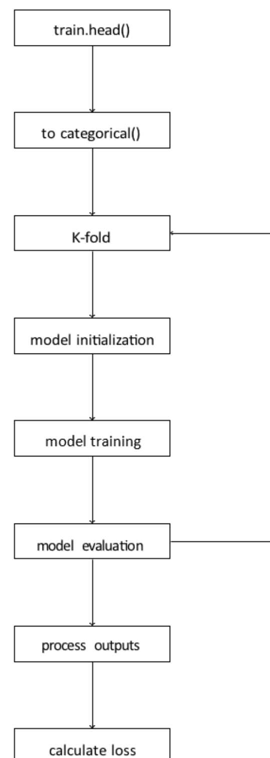

Fig. 7 Flowchart illustrating the key steps of the code execution process.

Results and Analysis

Our results are compared mainly based on one metric, which is the overall accuracy achieved by the model. This is obtained after running our trained model with the remaining dataset (in our case, 20% of the original dataset), and counting how many of our model’s predictions matched the dataset’s labels (of vulnerable or non-vulnerable). After running our model on our testing dataset we got an accuracy of 90.59%, which is a big improvement on the accuracy presented by the rest of the pretrained models on Table 2. We believe there are many possible reasons as to why this model proved to be more effective:

Hyperparameter Optimization

The extensive manipulation of the model’s hyperparameters allowed us to reach the settings required for the model’s superior performance. Through the fine-tuning of the model, we were able to align its knowledge with the target task of vulnerability classification, though we believe there is still room for further optimization.

Dataset Quality and Size

The fact that we used a real dataset rather than a synthetic one probably had a big impact on the accuracy, especially given that it was the same dataset that we used for testing. This aspect enhanced the model’s ability to generalize to real-world vulnerability instances and improved its performance in real-world scenarios. Additionally, this ensured that it was exposed to the complexities, variations, and nuances present in real-world vulnerabilities, allowing it to learn and adapt effectively.

Discussion

Our proposed method distinguishes itself by covering a discrete requirement. While some models may be particularly good at finding vulnerabilities within synthetic datasets, they often struggle with real-world data. In light of these limita-

 © The National High School Journal of Science 2023

Technique	Training	Testing	Accuracy
VulDeePecker	NVD/SARD	ReVeal	79.05
SySeVR	NVD/SARD		79.48
Russel et al.	Juliet		38.11
Russel et al.	Draper		70.08
Devign	FFMPeg+Qemu		66.24
Proposed Method	**ReVeal**		**90.59**

Table 2 Comparing our results (bottom row) to those presented by Chakraborty et al.[16] using pre-trained models on the ReVeal dataset.

tions, our model was specifically designed with the intention of being used within a real-world context, prompting us to allocate significant efforts toward achieving a high level of accuracy on the ReVeal dataset. This turned out to be very successful, as we were able to train a model that outperformed all of the compared techniques on the aforementioned dataset (Table 2).

On the other hand, a limitation that our model faces is a lack of explainability. As with most transformer models, our complex architecture makes it challenging to interpret and understand what patterns may lead to certain predictions. This can hinder the debugging process when trying to address the misclassification of an input and may prompt the need for additional validation mechanisms if it were to be implemented.

Conclusion

This paper demonstrates the advantages that come with utilizing transformers, more specifically the RoBERTa model, within a vulnerability detection context. We show that its implementation of attention is essential for enhanced accuracy using real-world data, even outperforming the previously compared pre-trained models. For researchers who aim to build upon this study, we recommend testing other types of neural network architectures within this context, using this same dataset for training and testing. Additionally, the study could benefit from testing using the same architecture but with different pre-trained transformer models, such as XLM or BERT. Moreover, we believe that our model's hyperparameters could be further optimized to get an even better efficiency.

Acknowledgements

I would like to acknowledge Lumiere Research Inclusion Foundation for its generous support in making this research possible. Furthermore, I am deeply grateful to Seemanta Saha for his continuous guidance and mentorship throughout both the experimental and writing aspects of this study.

References

1 W. Kang, B. Son and K. Heo, Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22, New York, NY, USA.

2 F. Wu, J. Wang, J. Liu and W. Wang, *Vulnerability detection with deep learning*.

3 T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Scao, S. Gugger, M. Drame, Q. Lhoest and A. Rush, *Huggingface's transformers: Stateof-the-art natural language processing*.

4 *2022 data breach report*, Identity Theft Center.

5 *Imperva*.

6 N. Ayewah, W. Pugh, D. Hovemeyer, J. Morgenthaler and J. Penix, *IEEE Software*, **25**, 22–29,.

7 T. Muske and U. Khedker, 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE, pp. 270–280,.

8 C. Chen, B. Cui, J. Ma, R. Wu, J. Guo and W. Liu, *Computers Security*, **75**, 118–137,.

9 J. Goel and B. Mehtre, *3rd International Conference on Recent Trends in Computing 2015*, **57**, 710–715,.

10 P. Zeng, G. Lin, L. Pan and T. Yonghang, *IEEE Access*, **10**, year.

11 A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser and I. Polosukhin, *Attention is all you need*.

12 G. Tang, L. Yang, L. Zhang, W. Cao, L. Meng, H. He, H. Kuang, F. Yang and H. Wang, *attention-based automatic vulnerability detection approach with ggnn*.

13 H. Hanif and S. Maffeis, 2022 International Joint Conference on Neural Networks (IJCNN, pp. 1–8,.

14 Dependable Software Engineering. Theories, Tools, and Applications - 7th International Symposium, SETTA 2021, Beijing, China, pp. 161–177,.

15 Google, *Google colab*, `https://colab.research.google.com/`,.

16 S. Chakraborty, R. Krishna, Y. Ding and B. Ray, *IEEE Transactions on Software Engineering*, **48**, 3280–3296,.

17 Y. Hamdaoui, *Text classification using transformers (pytorch implementation*.

18 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*.

19 J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*.

20 Z. Guo, L. Zhu and L. Han, 2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI, pp. 845–849,.

21 Y. Wang, Y. Sun, Z. Ma, L. Gao, Y. Xu and T. Sun, 2020 12th International Conference on Intelligent Human.