Issue Tagging

Aarefa Bhurka ● Shrinivas Sampath Muthupalaniyappan

1. Introduction

In today's contemporary era, software plays a significant role in all industries. It had grown vast enough that maintaining the software is more complicated than making it. Software maintenance is one of the critical roles in making software successful [5]. Why do we need supervision? The software developed by humans is not perfect; they tend to make mistakes, which create bugs and issues later after the software development. We need to update and resolve issues regularly to keep the software constant[5]. Software maintenance involves tasks to mitigate potential defects in the code and evolve it according to the users' emerging needs [1]. Thus, issue tracking systems are tools to efficiently support and manage the potential problems arising in software systems. However, due to the ample number of issues that are received, there is no way the developer has enough time to identify and resolve every issue on time. Since tens and thousands of problems are flying in a day, it is very labor-intensive to map each case [2,3]. Identifying the issues is one of the first steps to resolving the issue.

Therefore, several studies have taken a developer-centric approach by detecting how developers spend time identifying the issue types and how they analyze the issue at first sight. This led to the development of several systems like detecting the problem automatically using the Natural Language Processing technique. Still, even though that model was good enough to predict the balanced test data, it failed to predict the unbalanced test data [5]. In this context, recent studies have introduced an advanced deep learning model like the Neural Language model, BERT (Bidirectional Encoder Representations from Transformers), which uses state-of-the-art Transfer Learning Architecture [6].

In this paper, we will be presenting a good-performing machine learning model that will successfully tag the issues. For the performance comparison of our model, we will be using the fasttext model as a baseline [5]. The advantage of challenging the baseline model would be a positive effect on the developers, which helps them prioritize the tasks better. Also, it helps the developers to classify the reports; in other words, it allows them to filter, allowing them to perform more efficient issue-handling processes than in the previous model. Our objective is to increase the individual recall and precision of the target classes (Bug, Question, and Enhancement). The flow of the model proposed in this paper is shown in **Figure 1.**

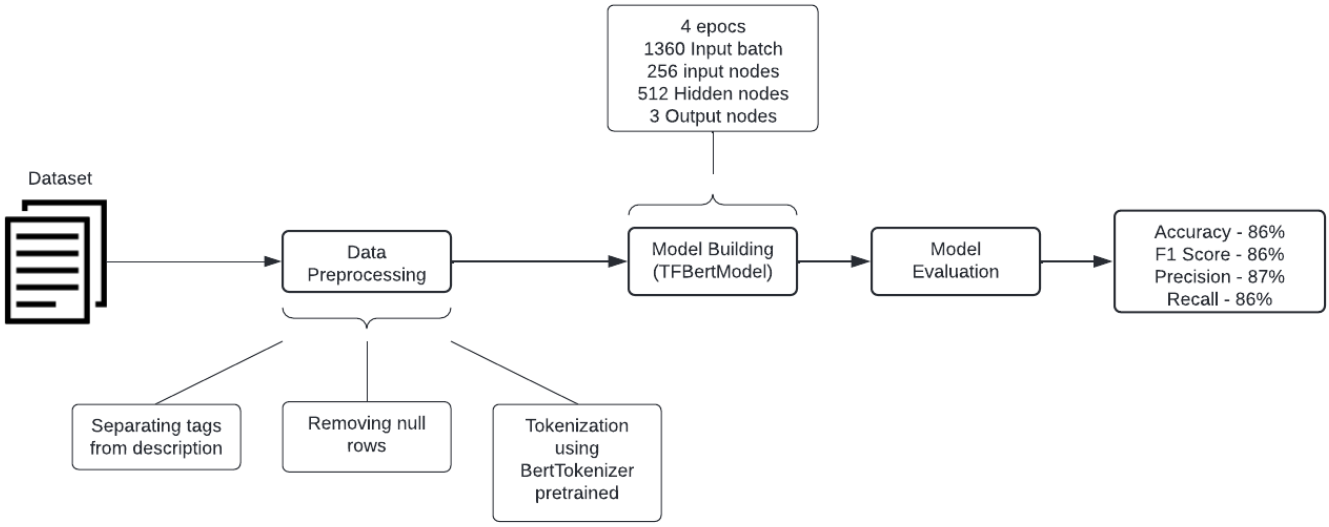

Figure 1 (Model Flow)

2. Approach and tool's overview:

This paper aims to perform text classification on the GitHub issues description and tag the issues with their respective categories, i.e., Bug, Enhancement, or Question. We will use an NLP model called "Bert." The BERT family of models uses the Transformer encoder architecture to process each token of input text in the full context of all tokens before and after, hence the name: Bidirectional Encoder Representations from Transformers. They compute vector-space representations of natural language suitable for use in deep learning models. BERT models are usually pre-trained on a large text corpus, then fine-tuned for specific tasks.

Preprocessing Model:

Data preprocessing is the crucial phase for any textual classification model to work. The first step of this phase is to separate class labels from the Description column, and then we drop the rows if there are any garbage values present in the issue description.

Furthermore, the text inputs need to be transformed to numeric token ids and arranged in several Tensors before being input to BERT. The BERT model comes with a transformer model pretrained on a large corpus of English data in a self-supervised fashion. The BertTokenizer object has a method called encode_plus from the BERT, which has many attributes that proved helpful in fine-tuning the tokens. We use a few details to tokenize the issue description: 1) truncation – sometimes a sequence may be too long for a model to handle. In this case, you will need to truncate the sequence to a shorter length. The parameter value is set to True to trim a sequence to the maximum size accepted by the model. 2) Padding is a strategy for ensuring tensors are rectangular by adding a special padding token to sentences with fewer tokens. We have set the padding parameter to max-length to pad the shorter sequences in the batch to match the most extended sequence. 3) add_special_tokens - Whether to encode the sequences with the special tokens relative to their model. We have set this parameter value as true to allow the Bert model to automatically identify when to add special tokens.

After passing the issue description text in the tokenizer, it returns a dictionary with three essential items: 1) input_ids are the indices corresponding to each token in the sentence. 2) attention_mask indicates whether a token should be attended to or not. 3) token_type_ids identify which sequence a token belongs to when there is more than one sequence. The target class density in the dataset after preprocessing is shown in Figure 2:

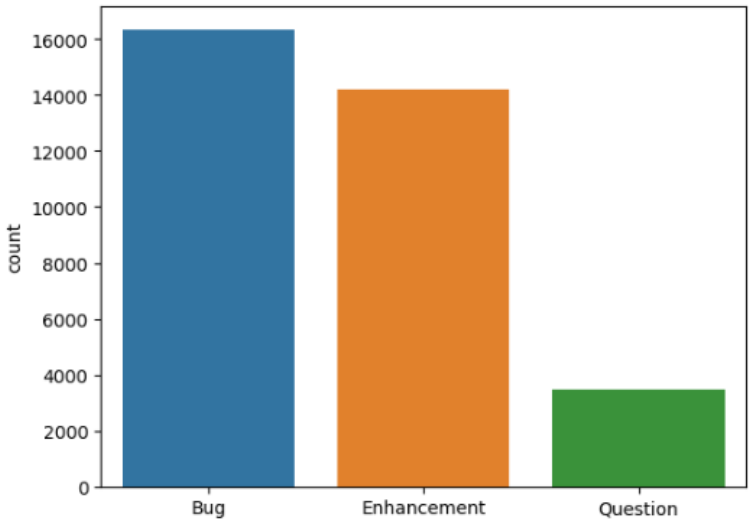

Figure 2 (Target class density)

Classification Model:

Motivation-

We have been working on replicating different research papers' model approaches and hyper tuning those models to classify the issues. But none of them was good enough to beat the performance of the fasttext mode (baseline model). After the popularity of BERT, researchers have tried to use it on different NLP tasks, and they were able to obtain state-of-the-art results. Because BERT is so powerful, fast, and easy to use for downstream tasks, it is likely to give promising results in multiclass classification. This became the primary motivation for pursuing this work using the BERT model.

Model Building-

Once the dataset is preprocessed, the next step is to train the model on the training dataset and evaluate its performance on the testing dataset.

BERT's model architecture is based on Transformers. It uses multilayer bidirectional transformer encoders for language representations. Based on the depth of the model architecture, two types of BERT models are introduced, namely $BERT_{Base}$ and $BERT_{Large}$. The $BERT_{Base}$ model uses 12 layers of transformers block with a hidden size of 768 and several self-attention heads as 12 and has around 110M trainable parameters. On the other hand, $BERT_{Large}$ uses 24 layers of transformers block with a hidden size of 1024 and number of self-attention heads as 16 and has around 340M trainable parameters. BERT uses the same model architecture for all the tasks, be it NLI, classification, or Question-Answering, with minimal changes such as adding an output layer for type. In this paper, we are using the $BERT_{Base}$ model for classification.

We are using PyTorch and the excellent PyTorch-Pretrained-BERT library for the model building. First, initialize an empty three-dimensional array for every row in our dataset. Then we change the values for the three-dimensional display by checking the target class, and we make that class one and the other class zero in a three-dimensional array. After this, the input of our machine learning model is made, and it consists of inputs id's, attention_mask, and labels for every line of a row in our dataset. The above three attributes will be fed into our final BERT model. Before providing the input in our BERT model, we split our 34000 rows of information into 16 batches or 16 input nodes. And for training, we are breaking the dataset into 80% training, and each node carries 80% of the data. The remaining 20% is used as the test data in our model, and out of the 80% in our training data, 20% is used as validation data.

Model Training and Optimization-

To pick up the suitable model for your classification problem, we evaluate the performance of five models and choose the best-performing model. The classification report of all five models is shown in Table 1.

The chosen model is the BERT base cased model, which does not change the alphabet cases in our dataset. Our dataset will be considered as it is. Using Keras, we create the object for input and initialize the shape of our information to be 256 for input ids and attention masks. After finalizing the form for our input, we pass both the input ids and attention masks to our input layer. The hidden layer is decided by giving the input layer to the

hidden layer, which has the shape of 512, and the activation layer used is RELU. Finally, we pass the hidden layer to the output layer, which has the form of 3 nodes, and the activation type used is SoftMax.
Then to make our model perform well, we chose ADAM because it controls the learning rate to the power of exponential, and it helps to prevent the decay rate of the learning rate. We are using the Keras library, and finally, accuracy, precision, recall, and F1score are determined by compiling and fitting our model.

Classifier	Precision	Recall	F1 Score	Accuracy
Random Forest	59%	47%	42%	46%
K Neighbors	62%	52%	47%	52%
SGD	68%	68%	68%	68%
Naïve Bayes	54%	53%	52%	53%
TFBertModel	87%	86%	86%	86%

Table 1 (Comparison of model performance)

Performance Evaluation:

In this section, we describe the baseline approach to assessing the classification performance of the BERT model. The goal of our experiment is twofold. On the one hand, we compare $BERT_{Base}$ with the baseline approach (Fasttext) to observe whether the text-based model can achieve comparable or better results than $BERT_{Base}$. For training and testing the Fasttext model, we need to convert our training and testing datasets to a text file for our fast text model to accept the input. The data is split for Fast text and $BERT_{Base}$ in 80:20 format. An unbalanced data set is used to compare both models. Well-known information retrieval metrics like precision, recall, and F-measure is used here to perform the evaluation.

The classification performance achieved by $BERT_{Base}$ and the baseline approach using an unbalanced dataset are shown in Tables 2 and 3. We can see that the $BERT_{Base}$ model outclasses the fasttext model in each class like bug, enhancement, and question. The F-measure of 0.88 for the $BERT_{Base}$ model is significantly higher than the Fasttext model, 0.76. Thus, the BERTBase model consistently outperforms the baseline approach (Fasttext) for all labels and precision, recall, and F-measure metrics.

Metrics	Bug	Enhancement	Question
Precision	0.79	0.73	0.44
Recall	0.72	0.74	0.53
F-measure	0.75	0.74	0.48

Table 2 (Bert Model Classification Report)

Metrics	Bug	Enhancement	Question
Precision	0.91	0.86	0.72
Recall	0.86	0.82	0.68
F-measure	0.88	0.84	0.70

Table 3 (Fasttext Model Classification Report)

3. Threats to validity

- Due to the limited availability of GPU, we could not test our model with epochs more than 4. Hence there are greater chances that increasing the number of periods will increase the mode's performance.
- Bert model has a vast network. Even though it is relatively more accurate than other classification models, we cannot ensure the time taken by this model to classify the enormous amount of data.

4. Conclusion:

We performed a T paired test comparing the Bert model to the base model and achieved a significant p-value (0.3985). The proposed model in this paper outperforms the fasttext. However, training this model over massive data will require ample time, depending upon the GPU availability. Future work will be aimed (i) at improving the training time equipped with high GPU, as well as (ii) at working with real-time data from the GitHub to analyze our model.

References:

[1] A. Di Sorbo, G. Grano, C. Aaron Visaggio, S. Panichella, Investigating the criticality of user-reported issues through their relations with app rating, J. Softw. Evol. Process (2020) e2316, https://onlinelibrary.wiley.com /doi /pdf /10 .1002 /smr.2316.6

[2] T.F. Bissyandé, D. Lo, L. Jiang, L. Réveillere, J. Klein, Y. Le Traon, Got issues? Who cares about it? A large-scale investigation of issue trackers from GitHub, in: International Symposium on Software Reliability Engineering, 2013, pp. 188–197.

[3] S. Panichella, G. Bavota, M.D. Penta, G. Canfora, G. Antoniol, how developers' collaborations identified from different sources tell us about code changes, in: ICSME, IEEE, 2014, pp. 251–260.
[4] J.L.C. Izquierdo, V. Cosentino, B. Rolandi, A. Bergel, J. Cabot, Gila: Github label analyzer, in: International Conference on Software Analysis, Evolution, and Reengineering, SANER, 2015, pp. 479–483.
[5] Rafael Kallis a, Andrea Di Sorbo b, Gerardo Canfora b, Sebastiano Panichella: Predicting issue types on GitHub, in Science of Computer Programming Volume 205, 1 May 2021, 102598.
[6] Manish Munikar, Sushil Shakya and Aakash Shrestha Fine-grained Sentiment Classification using BERT presented in 2019 Artificial Intelligence for Transforming Business and Society (AITB) IEEE, 2019.