BERT-Based GitHub Issue Report Classification

Mohammed Latif Siddiq
University of Notre Dame
Notre Dame, Indiana, USA
msiddiq3@nd.edu

Joanna C. S. Santos
University of Notre Dame
Notre Dame, Indiana, USA
joannacss@nd.edu

ABSTRACT

Issue tracking is one of the integral parts of software development, especially for open source projects. GitHub, a commonly used software management tool, provides its own issue tracking system. Each issue can have various tags, which are manually assigned by the project's developers. However, manually labeling software reports is a time-consuming and error-prone task. In this paper, we describe a BERT-based classification technique to automatically label issues as questions, bugs, or enhancements. Our approach classified reported issues with an F1-score of on average 0.8571 and of with highest 0.8586. Our technique performs better than the previous FastText based machine learning technique with an F1-score of 0.8162.

CCS CONCEPTS

• **Machine Learning** → *Natural Language Processing*; • **Software and Its Engineering** → *Software creation and management*.

KEYWORDS

datasets, text processing, GitHub issue, multi-classification, pre-trained model

ACM Reference Format:
Mohammed Latif Siddiq and Joanna C. S. Santos. 2022. BERT-Based GitHub Issue Report Classification. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software maintenance is a crucial part of the Software Development Life Cycle (SDLC) to mitigate vulnerabilities, fix bugs and evolve the software according to the users' needs [5, 16]. Issue Tracking Systems (ISTs) are frequently used during software development to aid software maintenance and evolution. These systems allow users to create new entries reporting a bug, requesting a new feature or asking questions regarding the project. Software engineers use the information provided in these entries to understand the nature of the report, and, in case of actual bugs, narrow down the list of files that are needed to be changed to fix the issue [19]. Engineers also use ISTs to track open issues, obtain additional information from reporters, and discuss potential bug-fixing solutions (including prioritizing issues/features to be developed).

GitHub, a widely used project management software, provides a built-in issue tracking system where users can ask questions, give suggestions on new features and point out possible bugs. Since these ISTs can be open to the public, developers need to triage new entries to understand the nature of the report (whether it is an actual valid bug, a feature or simply a question) to assign a custom label [3]. However, it can be difficult for developers, especially for a popular project, to manually label the issues. This manual process can be error-prone, labor-intensive, and time-consuming [6].

In this paper, we describe a BERT-based[1] model to predict an issue's type. We used a train set encompassing more than **700,000** labeled issue reports extracted from real open-source projects and a test set with **80,518** issues to evaluate our solutions provided by the organizers of the NLBSE'22 tool competition [7]. Our approach achieved highest **0.8586** as an F1-score, which exceeds the baseline model F1-score, **0.8162**. Our implementation is available on GitHub: https://github.com/s2e-lab/BERT-Based-GitHub-Issue-Classification.

This paper is organized as follows: Section 2 describes the approach to classify GitHub issue reports. Section 3 presents the results. Section 4 describes the current state-of-the-art in predicting GitHub issue reports. Finally, Section 5 concludes the paper with future directions.

2 APPROACH

In our work, we trained and tuned a multi-class classifier to classify GitHub issue reports. Figure 1 presents an overview of our approach. The following subsections discuss the dataset, the pre-processing and tuning steps, and the training and evaluation procedures.

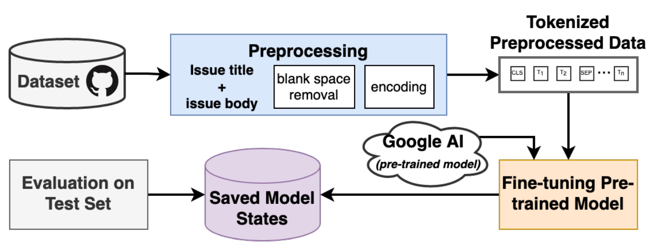

Figure 1: Overview of the Approach

2.1 Dataset

The dataset contains **803,417** labeled issue reports collected from real GitHub projects. Each entry contains the following metadata:

[1]BERT: Bidirectional Encoder Representations from Transformers.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

(1) **Label**: It indicates the nature of the report, which can be one of the following: *bug*, *enhancement*, and *question*.
(2) **Issue title**: A short descriptive sentence that indicates at a glance what the issue is about.
(3) **Issue body**: Each report must contain an issue body that includes a description explaining the purpose of the issue. This can consist of any details that might help resolve the issue.
(4) **Issue URL**: the URL to access the report on GitHub.
(5) **Repository URL**: Every issue is associated with a GitHub repository. This metadata stores the link of the remote GitHub repository.
(6) **Creation timestamp**: the timestamp of when the report was created.
(7) **Author association**: It describes how the issue creator is related to the repository. There are five types of author associations: *Owner*, *Collaborator*, *Contributor*, *Member*, *None*.

The dataset is split into a *training set* and a *test set* by the organizers of the NLBSE'22 tool competition [7]. The training set contains **722,899** (90%) and the test set contains **80,518** (10%) of the total labeled issue reports. We split the training set into new train and validation set. The new train set contains 85% of the previous train set, and 15% of the data is in the validation set. We used `train_test_split` function from scikit-learn [14] where we re-shuffled the data and split in a stratified fashion, using the class labels because our dataset is imbalanced. The dataset distribution is given in Table 1.

Set	Bug	Enhancement	Question	Total
Train set	306,937 (50.0%)	254,468 (41.4%)	53,059 (8.6%)	614,464
Validation set	54,166 (50.0%)	44,906 (41.4%)	9,363 (8.6%)	108,435
Test set	40,152 (49.9%)	33,290 (41.3%)	7,076 (8.8%)	80,518

Table 1: Dataset distribution.

2.2 Preprocessing

Before training the model, we processed the data and fixed the model's hyperparameters. This section discusses data processing steps and details of the hyper-parameters (i.e., Pre-trained Model, Optimizer, and Scheduler).

2.2.1 Text Cleaning & Feature Extraction. A GitHub issue contains a *title* and a detailed *description* of the issue. It may contain code segments or screenshots of the output. For example, Figure 2 shows a GitHub issue from Facebook's Flow repository containing code segments. Hence, we need to clean the data before encoding it.

As previously explained in Section 2.1, each issue has seven metadata in which one of them is the *label*. We consider the issue *title* and the issue *body* as the main features out of six features.

First, we concatenate these two features into new metadata, which we refer to as *issue data*. Then, we used the Gensim[2] library to remove repeating whitespace characters (i.e., spaces, tabs, and line breaks) from the *issue data*. Moreover, we replaced tabs and line breaks with spaces. This processed issue data is used as the feature of our model.

[2] https://radimrehurek.com/gensim/

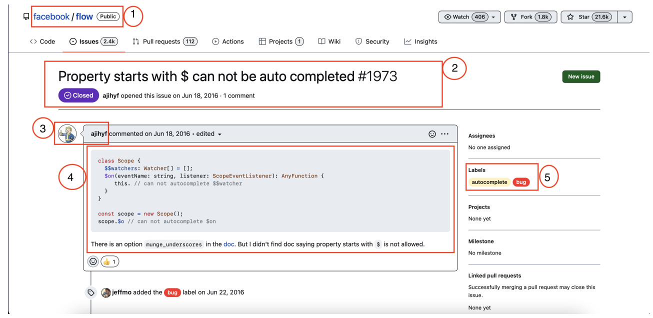

Figure 2: GitHub issue with bug tag from Facebook/flow 1. Repository Name, 2. Issue title, 3. Author handle and avatar, 4. Issue body, 5. Label

2.2.2 Encoding. To use a pre-trained BERT model, the feature data must be divided into *tokens*, then mapped to their respective indexes in the tokenizer vocabulary. We used `BertTokenizer`[3] implementation which is available in Hugging Face's transformers [18] package using the pre-trained model 'bert-base-uncased' [4]. This implementation tokenizes our issue data to feed into the pre-trained model. The 'bert-base-uncased' model is trained in English using a masked language modeling (MLM) objective. This model is uncased, which means it is case-sensitive. For instance, it differentiates between "issue" and "Issue."

BERT is a pre-trained model that requires input data in a specific format. Thus, we need the following items [11]:

- **[SEP]**: This token marks the end of a sentence or the separation between two sentences
- **[CLS]**: This token is used at the beginning of our text and is used for classification tasks, but BERT expects it no matter what your application is.
- **Token**: It complies with the fixed vocabulary used in BERT.
- **Token ID**: It is for the token generated from BERT's tokenizer.
- **Mask ID**: It indicates which elements in the sequence are tokens and which are padding elements.
- **Segment IDs**: It distinguishes different sentences.
- **Positional Embedding**: It shows token position within the sequence.

We used the `batch_encode_plus` method from BertTokenizer to handle the specific format described before. We used this method to extract the attention mask with the following parameters: `add_special_token` is set to ***true*** to encode special token, `return_attention_mask` is set to ***true*** to get the attention mask; `padding` is set to ***'longest'*** to pad to the longest sequence in the batch; `truncation` is set to ***true*** to truncate to the maximum acceptable input length for the model, and `return_tensors` is set to ***'pt'*** to get tensors of PyTorch `torch.Tensor` objects.

We used *token ids* (`input_ids`) and *attention mask* to create a `TensorDataset` which is later used into `DataLoader` with random sampling and batch size as four to train and evaluate the model. However, the authors of the BERT paper recommended using a batch size of 16 or 32 [4]. But our GPU has a memory limitation,

[3] https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertTokenizer

causing out-of-memory errors. We used a shorter batch size to solve this memory constrain at the cost of an increased training time.

2.2.3 Pretrained Model. We used BERT pre-trained model from Google AI [4], which has created a stir in the Machine Learning field by delivering cutting-edge findings in various NLP tasks, including Question Answering (SQuAD v1.1), Natural Language Inference (MNLI), and others. The use of the bidirectional training of Transformer [17], a prominent attention model, for language modeling is BERT's fundamental technological breakthrough. This contrasts with previous efforts, which looked at a text sequence from left to right or combined left-to-right and right-to-left training. [12]. The research findings suggest that bidirectionally trained language models can better understand language context and flow than single-direction language models.

We first alter the pre-trained BERT model to provide classification outputs in our work. Then, we keep training the model on our dataset until the complete model, end-to-end, is well-suited to our objective. We used BertForSequenceClassification[4] from Hugging Face. This is the standard BERT model with a single classification layer placed on top, which we employed as a document classifier. The pre-trained BERT model and the additional untrained classification layer get trained on our dataset. We utilized the pertained model "bert-base-uncased," which refers to the version with just lowercase characters ("uncased"). We do not want that the model returns attention weights and all hidden states. Hence, we disabled the flags while initiating the pre-trained model. After initiating the model, we fix our optimizer and scheduler.

2.2.4 Optimizer. We used AdamW [10] optimizer for the training. We used the implementation of the Adam algorithm with weight decay fix from HuggingFace. We used $1e^{-5}$ as the learning rate(`lr`) and $1e^{-8}$ as the `eps` which is a very small number to prevent any division by zero in the implementation.

2.2.5 Learning Rate Schedule. As the training develops, a learning rate schedule is a preset framework that changes the learning rate between epochs or iterations. We used a linear schedule with warmup implementation from HuggingFace. It created a schedule with a learning rate that decreases linearly from the initial `lr` set in the optimizer to 0, after a warmup period during which it increases linearly from 0 to the initial `lr` fixed in the optimizer.

2.3 Training

We used a previously created data loader to unpack the batch in our training, and each tensor was copied to the GPU. After clearing any previously calculated gradients, we performed a forward pass. In this step, the model provided the loss and logits, the output before the activation. Then, we perform a backward pass for calculating the gradients. We also clipped the norm of the gradients to 1.0. This is to help prevent the "exploding gradients" problem [15]. Then, we perform the optimizer's step to dictate how the parameters are modified based on their gradients, the learning rate, etc. Finally, we performed the scheduler's step to update the learning rate.

After each iteration, we calculated the average training loss and model performance on the validation set by calculating validation loss. We saved the model states for evaluating the model on training sets. Since the authors of BERT recommended having 2-4 iterations, we had **four iterations** for training.

2.4 Evaluation

We evaluate our model with the following metrics:

- **Precision (P)**: Precision (P) is calculated by dividing the number of records with correctly predicted labels by the total number of predicted observations in that class: $P = \frac{TP}{TP+FP}$. Here, TP is the number of records for which the label is predicted correctly, while FP denotes the number of records for which the label is incorrectly predicted. To compute the FP in multi-class classification, all observations that correspond to other labels and are mistakenly projected as group A are added for each group A.
- **Recall (R)**: Recall can be calculated for each group A by dividing the number of successfully predicted observations in A by the total number of observations in the corresponding class: $R = \frac{TP}{TP+FN}$. Here, FN is the number of observations in class A which are falsely predicted as other labels.
- **F1-Score (F1)**: The F1-score combines the precision and recall of a classifier into a single metric by taking their harmonic mean: $F1 = \frac{2.(P.R)}{P+R}$.

In our project, we calculated Precision (P), Recall (R), and F1-Score (F1) for each class. We used micro-averaging as the cross-class aggregation method to calculate global scores due to the class imbalance present in the data.

2.5 Implementation Details

We run our training job on a computing node that consists of Dual Twelve-core 2.2GHz Intel Xeon processors - 24 total cores, 128 GB RAM, and 4 NVIDIA GeForce GTX 1080 Ti GPU accelerators. We used a single core and one GPU to train our model. It takes approximately 18 hours to train for a single iteration. We used PyTorch [13] as a Deep learning framework ,the implementations of tokenizer, pre-trained model, optimizer, scheduler from HuggingFace[5], and evaluation matrices implementation from Scikit-learn [14].

3 RESULT

We used Bert pre-trained model to tune our dataset for classifying GitHub issue report into three classes. We compare our result with a baseline approach that used FastText [2]. Table 2 summarizes our results.

Our model consistently outperformed the baseline model using FastText. It also achieved a better result in every single class with respect to F1-score. It specially outperformed the baseline approach with respect to classifying issues that are labeled as *questions*.

We used micro-averaging as the cross-class aggregation method to calculate global scores. For that reason, Precision(P), Recall(R), and F1-Score(F1) have the same value where the highest value is 0.8586 and the average is 0.8571.

4 RELATED WORK

Kallis et al. [9] used FastText [2] to predict the types of GitHub issues by using issue titles and description as features. They built Ticket

[4] https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertForSequenceClassification

[5] https://huggingface.co/

Model	Matix	Bug	Enhancement	Question	Global
FastText	Precision	0.8314	0.8155	0.6521	0.8162
	Recall	0.8725	0.8464	0.3502	
	F1-Score	0.8515	0.8307	0.4557	
Epoch 1	Precision	0.8755	0.8486	0.7269	0.8554
	Recall	0.8913	0.8934	0.4708	
	F1-Score	0.8833	0.8704	0.5714	
Epoch 2	Precision	0.8831	0.8591	0.6789	**0.8586**
	Recall	0.8886	0.8880	0.5469	
	F1-Score	0.8859	**0.8733**	**0.6058**	
Epoch 3	Precision	0.8722	0.8573	0.7392	0.8584
	Recall	0.90146	0.8875	0.4740	
	F1-Score	**0.8866**	0.8721	0.5776	
Epoch 4	Precision	0.8763	0.8631	0.6706	0.8561
	Recall	0.8927	0.8772	0.5466	
	F1-Score	0.8844	0.8701	0.6023	

Table 2: Result Comparison Between FastText and our Bert-based Model.

Tagger, a GitHub app to help the developer to predict the types [8]. They achieved 0.75, 0.74 and 0.48 F1-score for bug, enhancement, and question label respectively by training on a balanced set and testing on a unbalanced set.

Artmann et al. [1] investigated the algorithms linear regression (LR), convolutional neural network (CNN), recurrent neural network (RNN), random forest (RF), and k-nearest-neighbor (KNN) - in multi-label text classification for GitHub issue reports. They used a dataset consisting of 38 thousand training rows and the test set has around 12 thousand rows. They splitted there dataset into three smaller datasets with different number of labels. The CNN has the highest F1-score for every data set and scored 0.4374, 0.4506, and 0.4390.

Fan et al. [6] studied text-based classification approaches on a large-scale dataset of GitHub issue reports. Four different machine learning classifiers i.e. Support Vector Machine(SVM), Naive Bayes, Logistic Regression, and Random Forest were evaluated on 80 popular projects in GitHub consisting of about 252,000 issues. They labelled the issues into two classes: bug and non-bug. They introduced a new matrix based on F1-score, average F-measure as f_{avg} , F-measure of bug (nonbug) as f_{bug} (f_{nonbug}), and number of bug (nonbug) as n_{bug}(n_{nonbug}).

$$f_{avg} = \frac{n_{bug} * f_{bug} + n_{nonbug} * f_{nonbug}}{n_{bug} + n_{nonbug}} \quad (1)$$

They achieved text-based classification approaches can achieve 69.7% to 98.9% of average F-measure (calculated as Equation 1) on their dataset, and the SVM classifier is the most effective approach compared to other typical classifiers.

5 CONCLUSION

Automated prediction of issue types can be very helpful in developing software, specially in open source projects where not only the repository's owner and moderators, but also users can report an issue. In this paper, we used a BERT-Based approach to automatically label issues as a question, bug or enhancement. Our model achieved an F1-Score of 0.8586, indicating that it can be used to predict issue report class to reduce manual work. In the future, we aim to improve our approach with a larger dataset, especially for issue reports with question tags. This approach can be integrated as a GitHub extension to have live-action.

REFERENCES

[1] Daniel Artmann. 2020. Applying machine learning algorithms to multi-label text classification on GitHub issues.

[2] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. arXiv:1607.04606 [cs.CL]

[3] Javier Luis Cánovas Izquierdo, Valerio Cosentino, Belén Rolandi, Alexandre Bergel, and Jordi Cabot. 2015. GiLA: GitHub label analyzer. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 479–483. https://doi.org/10.1109/SANER.2015.7081860

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[5] Andrea Di Sorbo, Giovanni Grano, Corrado Aaron Visaggio, and Sebastiano Panichella. 2021. Investigating the criticality of user-reported issues through their relations with app rating. *Journal of Software: Evolution and Process* 33, 3 (2021), e2316.

[6] Qiang Fan, Yue Yu, Gang Yin, Tao Wang, and Huaimin Wang. 2017. Where Is the Road for Issue Reports Classification Based on Text Mining?. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 121–130. https://doi.org/10.1109/ESEM.2017.19

[7] Rafael Kallis, Oscar Chaparro, Andrea Di Sorbo, and Sebastiano Panichella. 2022. NLBSE'22 Tool Competition. In *Proceedings of The 1st International Workshop on Natural Language-based Software Engineering (NLBSE'22)*.

[8] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. 2019. Ticket Tagger: Machine Learning Driven Issue Classification. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 406–409. https://doi.org/10.1109/ICSME.2019.00070

[9] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. 2021. Predicting issue types on GitHub. *Science of Computer Programming* 205 (2021), 102598. https://doi.org/10.1016/j.scico.2020.102598

[10] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101 [cs.LG]

[11] Chris McCormick and Nick Ryan. 2019. Bert word embeddings tutorial. https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/#2-input-formatting

[12] Oren Melamud, Jacob Goldberger, and Ido Dagan. 2016. context2vec: Learning Generic Context Embedding with Bidirectional LSTM. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*. Association for Computational Linguistics, Berlin, Germany, 51–61. https://doi.org/10.18653/v1/K16-1006

[13] Adam Paszke, Sam Gross, Francisco Massa, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, et al. 2018. Scikit-learn: Machine Learning in Python. arXiv:1201.0490 [cs.LG]

[15] George Philipp, Dawn Song, and Jaime G. Carbonell. 2018. The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions. arXiv:1712.05577 [cs.LG]

[16] S Shylesh. 2017. A study of software development life cycle process models. In *National Conference on Reinventing Opportunities in Management, IT, and Social Sciences*. 534–541.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[18] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[19] Thomas Zimmermann, R. Premraj, Jonathan Sillito, and Silvia Breu. 2009. Improving Bug Tracking Systems. 247 – 250. https://doi.org/10.1109/ICSE-COMPANION.2009.5070993