Issue Report Classification

Le Hoang Long[1*]

[1*]School of Information and Communications Technology, Hanoi University of Science and Technology, Dai Co Viet, Hanoi, Vietnam.

Corresponding author(s). E-mail(s): hoanglong1712@gmail.com;

Abstract

Accurate issue classification is essential for efficient software project management. However, inconsistencies in the labels assigned to issues can adversely affect the performance of supervised classification models.
NLP-based approaches and tools have been introduced to enhance the efficiency of software engineers, processes, and products by automatically processing natural language artifacts such as issues, emails, and commits.
We believe that the availability of precise tools is increasingly essential for improving Software Engineering (SE) processes. Two critical processes are (i) issue management and prioritization, and (ii) code comment classification, where developers need to understand, classify, prioritize, and assign incoming issues and code comments reported by end-users and developers.

Keywords: multi-label, classification, issue, GNN, KG, TransR, issue type classification, multi-class classification, text processing, software maintenance, pre-trained model

1 Introduction

Software maintenance is an essential aspect of the software development life cycle, aimed at mitigating vulnerabilities, fixing bugs, and evolving the software to meet users' needs. Issue Tracking Systems are commonly used to support software maintenance and evolution during development. These systems enable users to report bugs, request new features, or ask questions about the project. Software engineers utilize the information from these entries to understand the nature of the report and, in the case of actual bugs, identify the files that need changes to resolve the issue. Developers also use Issue Tracking Systems to monitor open problems, gather additional information from reporters, and discuss potential solutions, including prioritizing issues and features for development.

GitHub, a widely used project management tool, offers a built-in issue-tracking system where users can ask questions, suggest new features, and report possible bugs. Since these systems can be publicly accessible, developers must triage new entries to determine whether they are valid bugs, feature requests, or questions, and assign appropriate labels. However, manually labeling issues can be challenging, error-prone, labor-intensive, and time-consuming for popular projects.

This paper presents a BERT-based model designed to predict the type of an issue. This dataset was provided by the organizers of the NLBSE'24 tool competition [1] [2]. Our implementation can be found on GitHub: https://github.com/hoanglong1712/Dai-Hoc-Bach-Khoa-Ha-Noi/tree/main/Seminar%202/Project%201%3A%20Issue%20Report%20Classification.

The paper is structured as follows: Section 2 details the approach to classifying issue reports, Section 3 presents the results, Section 4 discusses the current state-of-the-art in predicting issue reports, and Section 5 concludes with future directions. [3],[4], [5].

2 The architecture and details of the classification models

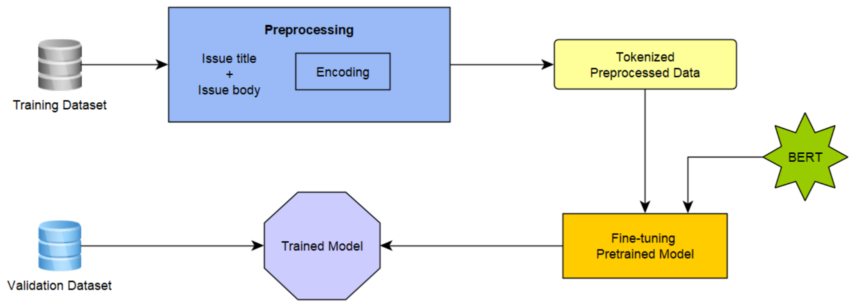


Fig. 1 Overview Approach

We developed and fine-tuned a multi-class classifier to automatically label report issues. Figure 1 provides an overview of our methodology. The subsequent subsections cover the dataset, pre-processing, and tuning steps, as well as the training and evaluation procedures.

2.1 Dataset

The dataset comprises 1.5 thousand labeled issue reports extracted from 5 real open-source projects. The training set is balanced, i.e., it contains the same number of issues for each class. Each entry includes the following metadata:

1. Label: Indicates the nature of the report (bug, enhancement, question).
2. Repository: Link to the associated GitHub repository.
3. Title: A brief description of the issue.
4. Body: Detailed description explaining the issue.

Each issue is labeled with one class that indicates the issue type, namely, bug, feature, and question. Issues with multiple labels are excluded from the dataset. The dataset is then split into a training set (50%) and a test set (50%).

2.2 Pre-processing

Before training the model, we preprocessed the data and set the model's hyperparameters. This section covers the data preprocessing steps and the hyperparameter details.

2.2.1 Text Cleaning & Feature Extraction

An issue includes a title and a body, which may contain code segments or screenshots. Therefore, we need to clean the data before encoding it.

As explained in Section 2.1, each issue has four metadata fields, one of which is the label. We consider the issue title and body as the primary features.

First, we combined the title and body into a new metadata field called issue data. We then removed repeating whitespace characters (spaces, tabs, and line breaks) and replaced tabs and line breaks with spaces. This processed issue data is used as the feature for our model.

Second, we generate a list of labels for each instance "'facebook/react', 'tensorflow/tensorflow', 'microsoft/vscode', 'bitcoin/bitcoin', 'opencv/opencv', 'bug', 'feature', and 'question'", which can be represented in binary format (1 for presence, 0 for absence).

2.2.2 Encoding

To utilize a pre-trained BERT model, the feature data needs to be tokenized and mapped to their corresponding indexes in the tokenizer's vocabulary. We employed the BertTokenizer from Hugging Face's transformers package, specifically using the bert-base-uncased pre-trained model to tokenize our issue data. This model, trained in English with a masked language modeling (MLM) objective, is case-insensitive, meaning it treats "data" and "DATA" as the same token.

BERT is a pre-trained model that requires input data to be formatted in a specific way. The necessary components include[6]:

- $[SEP]$: This token signifies the end of a sentence or the separation between two sentences.

- $[CLS]$: This token is placed at the beginning of the text and is used for classification tasks, though BERT expects it regardless of the application.
- Token: This adheres to the fixed vocabulary used by BERT.
- Token ID: This is the identifier for the token generated by BERT's tokenizer.
- Mask ID: This indicates which elements in the sequence are tokens and which are padding elements.
- Segment IDs: These distinguish different sentences.
- Positional Embedding: This shows the position of the token within the sequence.

We utilized token IDs (input_ids) and attention masks to construct a Tensor Dataset, which was subsequently fed into a DataLoader for model training and evaluation. The DataLoader was configured with random sampling and a batch size of 64. Although the authors of the BERT paper recommended a batch size of 16 or 32, smaller batch sizes were insufficient for capturing features from the data. Therefore, we opted for a larger batch size to address this underfitting issue.

2.2.3 Pretrained Model

We employed the BERT pre-trained model developed by Google AI [7]. BERT's fundamental technological breakthrough lies in its bidirectional training of Transformers for language modeling. This approach contrasts with previous methods that processed text sequences either from left to right or by combining left-to-right and right-to-left training. Research findings indicate that bidirectionally trained language models have a superior ability to understand language context and flow compared to single-direction models.

We initially modified the pre-trained BERT model to generate classification outputs for our work. Subsequently, we continued training the model on our dataset until the entire model, end-to-end, was well-suited to our objective. This involved using the standard BERT model with a single classification layer added on top, which we employed as a document classifier. Both the pre-trained BERT model and the additional untrained classification layer were trained on our dataset. We utilized the "bert-base-uncased" pre-trained model, which operates with lowercase characters ("uncased"). To avoid returning attention weights and all hidden states, we disabled the corresponding flags during the initialization of the pre-trained model. After initializing the model, we configured our optimizer and scheduler as detailed in the subsequent sections.

2.2.4 Optimizer

We employed the AdamW optimizer for training, utilizing the implementation of the Adam algorithm with a weight decay fix from Hugging Face. We set the learning rate (lr) to $(1 \times 10^{-6} \times 3)$, which contributed to improved model accuracy.

2.3 Training

During the training phase, we utilized a pre-existing data loader to unpack the batch, transferring each tensor to the GPU. After clearing any previously calculated gradients, we executed a forward pass, during which the model produced the loss and logits as outputs prior to activation. Subsequently, we performed a backward pass to calculate the gradients.

Next, we executed the optimizer's step to adjust the parameters based on their gradients. After each iteration, we calculated the average training loss and assessed model performance on the validation set by computing the validation loss. We saved the model states for subsequent evaluation on the training sets. Given the learning rate of $(1 \times 10^{-6} \times 3)$, we conducted 70 iterations for training.

2.4 Evaluation

We evaluate our model using the following metrics by computing the Precision, Recall, and F1-Score for each class.

1. Here, TP (true positives) represents the number of records correctly predicted, while FP (false positives) denotes the number of records incorrectly predicted. FN (false negatives) is the number of observations in class A that are falsely predicted as other labels.
2. Precision (P): This is calculated by dividing the number of correctly predicted labels by the total number of predicted observations in that class:

$$P = \frac{TP}{TP + FP}$$

3. Recall : This is computed for each group A by dividing the number of successfully predicted observations in A by the total number of observations in the corresponding class:

$$R = \frac{TP}{TP + FN}$$

4. F1-Score (F1): The F1-score combines the precision and recall of a classifier into a single metric by taking their harmonic mean.

$$F1 = \frac{2 \times P \times R}{P + R}$$

2.5 Implementation Details

We conducted our training on Kaggle, utilizing a setup with virtual CPUs, 29 GB of RAM, and T4 GPU $\times 2$ accelerators. Our model was trained using dual GPUs, with each iteration taking approximately 10 minutes. We employed PyTorch [8] as our deep learning framework, incorporating the tokenizer, pre-trained model, optimizer, and scheduler from HuggingFace. For evaluation metrics, we utilized implementations from Scikit-learn[9].

3 Results

Table 1 Validation Results

Label	Repository	Accuracy	Precision	Recall	F1
bug	facebook/react	95%	0.9700	0.9749	0.9724
bug	tensorflow/tensorflow	90%	0.9450	0.9545	0.9497
bug	microsoft/vscode	70%	0.7800	0.8211	0.8000
bug	bitcoin/bitcoin	80%	0.7950	0.8281	0.8112
bug	opencv/opencv	76%	0.8450	0.8756	0.8601
feature	facebook/react	78%	0.8900	0.9082	0.8990
feature	tensorflow/tensorflow	80%	0.9000	0.9091	0.9045
feature	microsoft/vscode	81%	0.8900	0.9223	0.9059
feature	bitcoin/bitcoin	81%	0.9050	0.9235	0.9141
feature	opencv/opencv	80%	0.9000	0.9091	0.9045
question	facebook/react	72%	0.7900	0.8144	0.8020
question	tensorflow/tensorflow	85%	0.9150	0.9482	0.9313
question	microsoft/vscode	71%	0.8400	0.8660	0.8528
question	bitcoin/bitcoin	68%	0.7300	0.7644	0.7468
question	opencv/opencv	70%	0.8000	0.8466	0.8226
overall	overall	81%	0.8596	0.8850	0.8721

Our model achieved an average F1-Score of 0.8721, demonstrating its capability to predict issue report classes and thereby reduce manual effort. Moving forward, we plan to enhance our approach by utilizing a larger dataset, particularly focusing on issue reports with question tags.

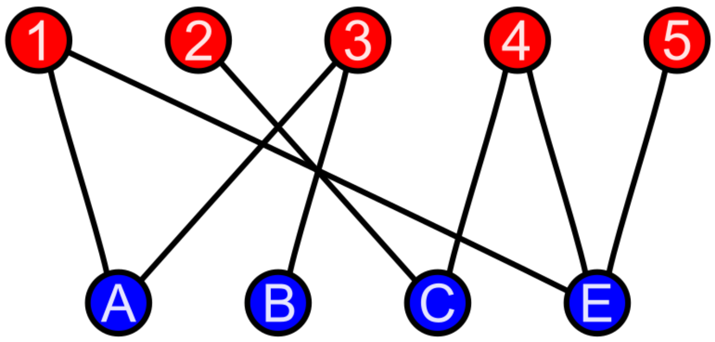
4 Future work

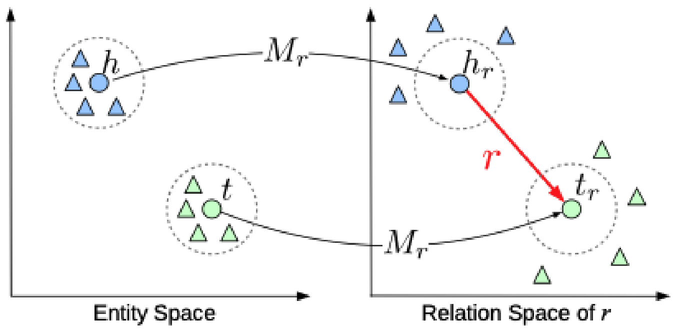
The current model functions, but it struggles with large-scale datasets. Training it on even a small dataset requires significant time and computing resources, making it impractical for real-world classification tasks.

Knowledge graph is an effective approach in this scenario. We could see that the dataset consists of two types of items: labels and data. The data is in text format, and there are eight different labels. We can construct a bipartite graph (or bigraph) based on the dataset. Bigraph is a graph whose vertices can be divided into two disjoint and independent sets and, that is, every edge connects a vertex into one in.

Knowledge graph completion focuses on link prediction between entities. In future work, we plan to explore knowledge graph embeddings. We are going to use TransR to create separate embeddings for entities and relations in their respective spaces. Then, we learn these embeddings by projecting entities from their space into the corresponding relation space and establishing translations between the projected entities [10], [11].

We apply a new method that models entities and relations in separate spaces specifically, an entity space and multiple relation-specific spaces. This approach performs translations within the corresponding relation space.





The fundamental concept of TransR is depicted in the figure above. For each triple (h, r, t), the entities in the entity space are first projected into the r-relation space as h_r and t_r using the operation M_r. This leads to the approximation $h_r + r_r \approx t_r$. The relation-specific projection brings together head and tail entities that share the relation (represented as colored circles) while distancing them from those that do not (represented as colored triangles).

In TransR, for each triple (h, r, t), the entity embeddings are represented as $h, t \in R^k$, while the relation embeddings are denoted as $r \in R^d$. It's important to note that the dimensions of the entity and relation embeddings do not have to be the same, meaning $k \neq d$.

For each relation r, we define a projection matrix $M_r \in R^{k \times d}$ that projects entities from the entity space into the relation space. Using this mapping matrix, we define the projected vectors of the entities as follows:

$$h_r = hM_r, t_r = tM_r$$

The score function is correspondingly defined as

$$f_r(h,t) = \| h_r + r - t_r \|_2^2$$

We define the following margin-based score function as the objective for training

$$L = \sum_{(h,r,t)\in S} \sum_{(h',r',t')\in S'} max(0, f_r(h,t) + \gamma - f_r(h',t'))$$

where max(x, y) aims to get the maximum between x and y, γ is the margin, S is the set of correct triples, and S' is the set of incorrect triples.

5 Conclusion

Automating the classification of issue types is particularly advantageous for software maintenance, especially in open-source projects with numerous user-reported issues. This paper explores a BERT-based method to automatically categorize issues as questions, bugs, or enhancements. Our model achieved an average F1-Score of 0.8721, demonstrating its capability to predict issue report classes and thereby reduce manual effort. Moving forward, we plan to enhance our approach by utilizing a larger dataset, particularly focusing on issue reports with question tags

References

[1] Kallis, R., Colavito, G., Al-Kaswan, A., Pascarella, L., Chaparro, O., Rani, P.: The nlbse'24 tool competition. In: Proceedings of The 3rd International Workshop on Natural Language-based Software Engineering (NLBSE'24) (2024)

[2] Al-Kaswan, A., Izadi, M., Van Deursen, A.: Stacc: Code comment classification using sentencetransformers. In: 2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE), pp. 28–31 (2023)

[3] Kallis, R., Di Sorbo, A., Canfora, G., Panichella, S.: Predicting issue types on github. Science of Computer Programming **205**, 102598 (2021) https://doi.org/10.1016/j.scico.2020.102598

[4] Kallis, R., Di Sorbo, A., Canfora, G., Panichella, S.: Ticket tagger: Machine learning driven issue classification. In: 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019, pp. 406–409. IEEE, ??? (2019). https://doi.org/10.1109/ICSME.2019.00070

[5] Pascarella, L., Bacchelli, A.: Classifying code comments in java open-source software systems. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) (2017). IEEE

[6] Chris McCormick, N.R.: In: Bert Word Embeddings Tutorial. https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/#2-input-formatting

[7] Jacob Devlin, K.L. Ming-Wei Chang, Toutanova., K.: In: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. CoRR Abs/1810.04805 (2018), vol. 26. arXiv (2018). https://arxiv.org/abs/1810.04805

[8] Adam Paszke, F.M.e.a. Sam Gross: In: PyTorch: An Imperative Style, High-Performance Deep Learning Library (2019). http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-highΘperformance-deep-learning-library.pdf

[9] Fabian Pedregosa, A.G.e.a. Gaël Varoquaux: In: Scikit-learn: Machine Learning in Python (2018). https://arxiv.org/abs/1712.05577

[10] Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., Yakhnenko, O.: Translating embeddings for modeling multi-relational data. In: Burges, C.J., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems, vol. 26. Curran Associates, Inc., ??? (2013). https://proceedings.neurips.cc/paper$_{f}iles/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-$ $Paper.pdf$

[11] Yankai Lin, M.S.Y.L.X.Z. Zhiyuan Liu: In: Learning Entity and Relation Embeddings for Knowledge Graph Completion. https://linyankai.github.io/publications/aaai2015_transr.pdf