The GitHub Issue Taxonomy

Zaara Zabeen Arpa
200042101
zaarazabeen@iut-dhaka.edu

Sadnam Sakib Apurbo
200042135
sadnamsakib@iut-dhaka.edu

Nazia Karim Khan Oishee
200042137
naziakarim@iut-dhaka.edu

Abstract

The GitHub Issue Taxonomy automates the categorization of GitHub issues into bugs, features, or questions using natural language processing. It enhances issue management and helps development teams prioritize tasks effectively, improving software development efficiency and user satisfaction. In our project, we have used BERT model for issue classification. To train and evaluate our model, we have used a data set sourced from Kaggle obtaining an accuracy score of 73%. Our models and code are available on GitHub at GitHub-Issue-Taxonomy.

1 Introduction

GitHub Issues are items that are created in a repository to plan, discuss and track work. Issues are widely used by developers to track work, give or receive feedback, collaborate on ideas or tasks, and efficiently communicate with others.These issues encompass a variety of categories including bugs, deadlines, features, questions, and more.GitHub Issues are widely used across various software development methodologies, including popular frameworks like Kanban and Agile. Recognizing the significance of Issues in software development methodologies, accurately determining the specific type to which an Issue belongs to becomes even more significant.

2 Problem Statement

Project managers often label the issues manually. Categorizing the labels manually is time-consuming and susceptible to errors. It can lead to inefficiencies and emergence of additional problems. Determining the accurate labels of issues is imperative in this field.

3 Solution

Our primary objective was to use advanced Natural Language Processing (NLP) techniques to classify the type of a given issue. After researching and exploring we came to the decision of using BERT (Bidirectional Encoder Representations from Transformers) model for text classification.

4 Background Study

BERT is a pre-trained model on large data sets unlike RNN which is not pre-trained. Therefore it has been observed that for complex tasks that requires an understanding of context, relationships, and semantics, BERT tends to perform well. BERT can capture bi-directional context of speech or sentence by considering both left and right context of each word. Whereas, deep neural network like RNN can capture context in unidirectional way. BERT is an encoder-only transformer architecture which contains a stack of encoders. Each of these encoders include a feed forward neural network and capture the importance of words sequentially. BERT processes the input through the stack of the encoders in forward propagation and calculate the gradient of the loss in backward propagation. After evaluating these characteristics and advantages, we opted to choose BERT.

5 Model

Bidirectional Encoder Representations from Transformers (BERT) is a language model based on the transformer architecture. BERT was originally implemented in the English language at two model sizes. For our project we have used BERTBASE which has 12 encoders and each of these encoders consists of 768 feed-forward networks.

We downloaded the pre-trained BERT models from Hugging Face's Transformers library. Hugging Face Transformers is an open-source framework which provides a wide range of pre-trained models including different variants of BERT.

6 Data

The data set is collected from Kaggle.We had one Train.json file with 150k samples, one Test.json file with 30k samples and one Train_extra.json with 300k samples.
Our data set contained three attributes - Title, Body and Label. The data set had three class labels- Bug, Feature and Question.

7 Implementation

7.1 Reading Datasets

pd.read_json() is a pandas function that reads data from a JSON file into a data frame. Here three data frames such as training set, extended training set and testing set which has been imported from the Datasets library.

7.2 Concatenation of Training Data

From the available two training sets,data has been concatenated using the pd.concat(). The concatenation has been done vertically (axis=0).

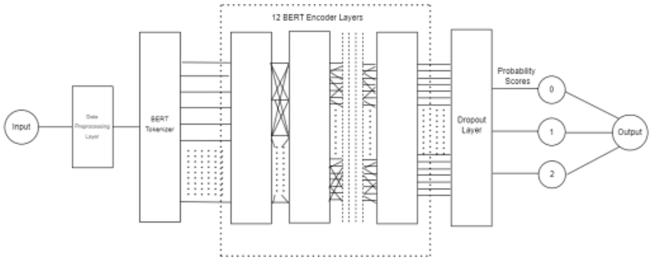

Figure 1: Neural Network Architecture

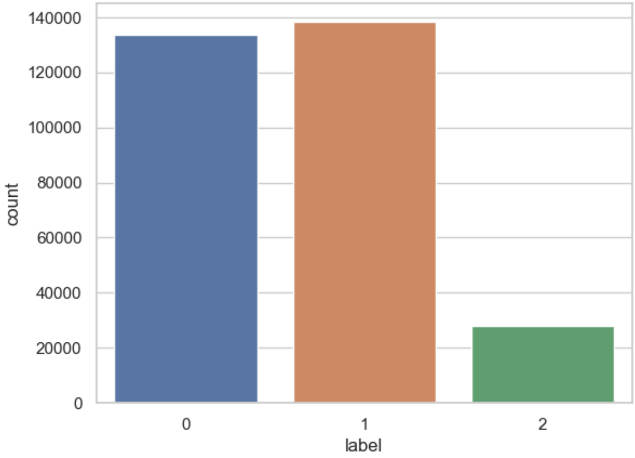

Figure 2: Distribution of Data

7.3 Down sampling of Training Data

Initially the data frames were unbalanced. We balanced it by down sampling.After down sampling each label contained 42201 samples.Balancing the data set was crucial to avoid biasness of the model.

7.4 Pre-processing of Training Data

The two columns of the main data set-’title’ and ’body’ are concatenated into a single line and placed in a new column named ’text’ in the data frame. The ’text’ column has been cleaned by removing the texts within square brackets, urls, html tags, punctuations, newlines and words containing digits. To clean the data set stopwords such as conjuctions, prepositions, punctuations etc are also removed.We used nltk ,re and string libraries for this task.

7.5 Class’ Creation

We created a ’config’ class to define the configuration parameters so that these parameters are consistent through out our code. Another class is ’Git Message’ class which is a data set class. The method of this class retrieves the Git commit message and its corresponding label at the given index, uses the BERT tokenizer to encode the Git commit message. It returns: ’input_ids’,’attention_mask’ and ’label’.

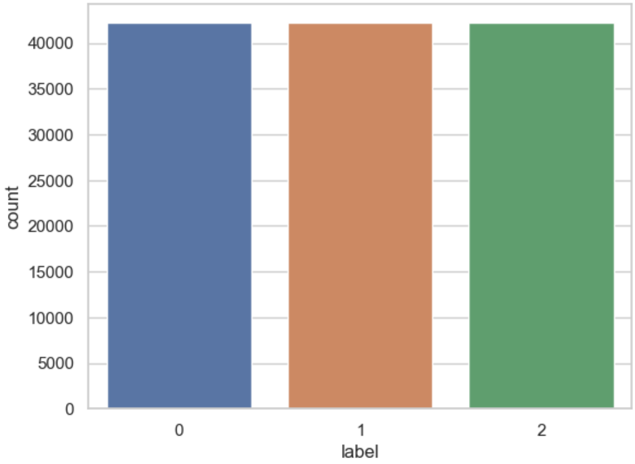

Figure 3: Down sampled Training Data Distribution

7.6 Data Loader

Data Loader takes four parameters: the input data, a tokenizer object used to tokenize the text data, the maximum length of tokenized text and the batch size for the data loader. It creates an instance of the ’Git_Message’ and creates a data_loader which returns the converted git message. Three instances of the loader for training, testing and validation data set are created .

7.7 PyTorch Neural Network modules

We have created a ’Bug Predictor’ class which is a subclass of PyTorch Neural Network modules.It uses the pre-trained model ’bert-based-uncased’. We have used a dropout layer as a regularization technique to prevent over fitting.

7.8 Optimizer

We have used ’AdamW’ optimizer to optimize the parameters.We found out that AdamW is the more appropriate optimizer for the given task.

7.9 Learning Rate

The learning rate is set to 1e-5 and no bias correction is applied.

7.10 Loss Function

We have selected cross-entropy loss function aka log loss as our loss function which measures the performance of the model.Its output is a probability value between 0 and 1. Pytorch’s cross entropy loss function is equivalent to the traditional ”Categorical Cross Entropy Loss Function”.

7.11 Training the model

For each of the batches in the data loader- the input ids, attention mask and target labels are moved to the specified device.In our case we have taken advantage of GPU CUDA cores which significantly increased training speed. A forward pass is performed of the model to obtain predictions for the given batch. The class labels are predicted and the loss are computed using the loss function, model’s prediction and target labels. The gradient of the loss is computed using 'loss.backward' with respect to the model parameters.

7.12 Evaluating the model

Evaluation is done by using eval() method that sets the model to evaluation mode. The batch iteration is done without calculating the gradients during the evaluation using 'with torch.no_grad()' with no optimization, scheduler and no gradients.

7.13 Final Prediction

With our model pipeline ready we can take an input to our 'predict_git_category' method which encodes the message and uses the model to predict the category(bug, feature or question) of the message.BERT tokenizer is used to encode the input git message by including tokenization and formatting operations such as truncation, padding, and returning tensors in PyTorch format. 'input_ids' and 'attention_mask' extracts the input token IDs and attention mask from the encoded messages and moves it to the specified device. After performing a forward pass using the pre-trained BERT model to obtain the outputs for each class, class name based on the index from the model's output is determined.

8 Test Accuracy

After training and evaluating the model for 10 epochs, we obtained a test accuracy of 73%.

9 Observation

After experimenting with several combinations of hyper parameters as presented below, we realized that AdamW is a more appropriate optimizer than SGD in terms of NLP tasks. Also we found out that increasing the dropout percentage gives us a better test accuracy as it reduces over fitting issues in models.

Size	**Optimizer**	**Dropout**	**Test Accuracy**	**Train Accuracy**
3000(2.5%)	AdamW	0.0	72.67%	64%
3000(2.5%)	SGD	0.0	33.6%	35%
3000(2.5%)	AdamW	0.2	74%	62.7%
3000(2.5%)	RMSProp	0.0	72%	59%

Table 1: Model Training Results

	Precision	Recall	F1-Score	Support
Class 0	0.75	0.74	0.75	4287
Class 1	0.74	0.74	0.74	4216
Class 2	0.69	0.71	0.70	4157
Accuracy			0.73	12660
Macro Avg	0.73	0.73	0.73	12660
Weighted Avg	0.73	0.73	0.73	12660

Table 2: Classification Scores

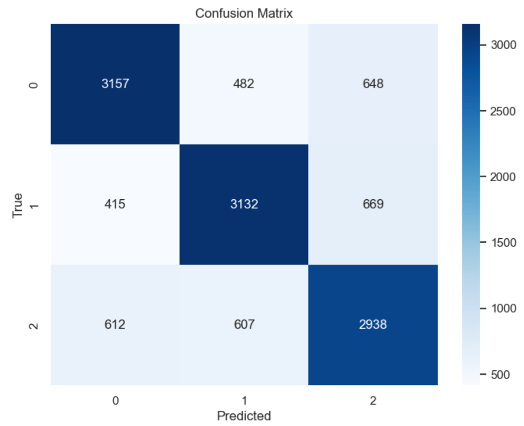

Figure 4: Confusion Matrix

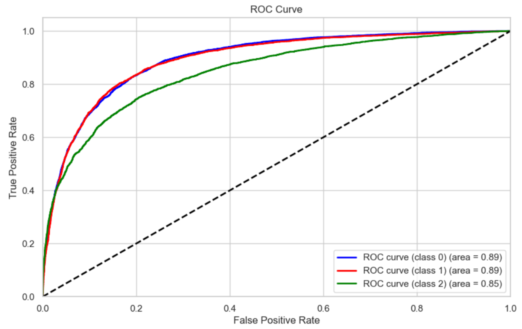

Figure 5: ROC Curve

10 Classification Report

Upon training and evaluating the model, we obtained F1-Score of 75%, 74% and 70% for Class 0,1 and 2 respectively. The accuracy was 73%. Precision, recall and other classification scores are given at Table 2. Figure 4 and 5 represents the Confusion matrix and ROC curve respectively.

References

- Predicting issue types on GitHub
- The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning)
- What is BERT?
- Text Classification with BERT in TensorFlow and PyTorch