



**UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO**

Department of Computer Science
MASTER'S DEGREE IN COMPUTER SCIENCE
ARTIFICIAL INTELLIGENCE CURRICULUM

ISSUE REPORT CLASSIFICATION USING BERT

Thesis in Software Engineering for AI-Enabled Systems

Supervisors:
Prof. Filippo Lanubile
Prof. Nicole Novielli

Graduating Candidate:
Giuseppe Colavito

19 July 2022

Table of Contents

1	Introduction	3
1.1	Thesis structure	4
2	Background	5
2.1	Issue Tracking	5
2.2	Text Classification	9
2.3	Pretrained Language Models	10
2.3.1	Transformers and Attention	10
2.3.2	BERT	14
3	Issue Report Classification	16
3.1	Related work	16
3.2	Challenge description	18
3.2.1	Dataset	19
4	Methodology	22
4.1	Research questions	22
4.2	Pre-processing	22
4.3	Model fine-tuning	23
4.4	Training the Issue Classifiers	25
4.5	Evaluation	28
5	Results	29
5.1	Comparison	29
6	Discussion	33
6.1	Error Analysis	33
6.1.1	Error Examples	34
6.2	Handling noise	37
7	Conclusions	41
7.1	Future Works	41
	List of Figures	43

List of Tables	44
References	45

1. Introduction

Automatic classification of issue types is crucial to support effective issue management and prioritization. Software developers, testers and customers routinely submit issue reports to software issue trackers to record the problems they face in using a software. The issues are then directed to appropriate experts for analysis and fixing. However, submitters often misclassify an improvement request as a bug and viceversa. This costs valuable developer time. The person filing the issue may not always make a fine-grained distinction between the different kinds of reports and instead file them as bugs only. In fact, research shows misclassifications are commonplace [1, 2]. In an elaborate study involving more than 7000 issues spanning 5 projects, researchers found that 33.8% of all reports are misclassified [2]. The consequence of misclassification could be costly: developers must spend their precious time to look into the reports and relabel them correctly. This is necessary also to understand which team should take charge of the issue. Hence it is worthwhile to explore if this classification can be done automatically, since it would be of great practical utility. [3].

Previous studies have proposed supervised approaches to address the task of automatically predicting the label to assign to a new issue. Early studies leveraged traditional machine learning, such as decision trees, naive Bayes classifiers, and logistic regression in combination with text-based features, achieving performance between 77% and 82% of accuracy [1]. More recently, researchers started to use deep learning and, in particular, for natural language processing, pre-trained language models, such as BERT and its variants [4, 5].

Kallis et al. [6, 7] proposed Ticket Tagger, which automatically predicts the labels to assign to issues at write time, with the aim of facilitating the issue management and prioritization processes. Ticket Tagger is a machine learning classifier that predicts the label to assign to issues trained on GitHub data. Specifically, Ticket Tagger leverages only the textual content of an issue title and body, whose vectorial representation is based on *fastText* [8], an open-source tool released by Facebook AI research.

In this work, we describe the systems we developed to participate in the tool competition of NLBSE'22 on automatic labeling of GitHub issues. The goal of the NLBSE'22 tool competition [9] is to build a classifier for automatic issue report classification. Ticket

Tagger [6, 7] is identified by the challenge organizer as the baseline system and all participants are invited to compare the performance of the proposed system with it ($F1 = .8591$). The organizers provided a dataset including more than 800K GitHub issue reports labeled as either *bug*, *enhancement*, or *question*, in line with the intent of the author of the issue [6, 7]. All issues are extracted from real open-source projects. The participants were invited to use the dataset to train and evaluate an approach for automatic classification of the three issue types in the dataset:

- bug,
- enhancement,
- question.

Inspired by recent advances in distributional semantics [10, 11, 12], we aim at assessing to what extent the text information only could be used to enhance the state of the art in automatic issue labeling. Specifically, we propose and evaluate two different approaches based on supervised learning that leverage the information available at the time of writing, that is the title and body of the issue and the issue-author association relation (e.g., collaborator, owner, etc.). We experiment with fine-tuning of BERT [10], a task-agnostic pre-trained language-model released by Google, and its variants ALBERT [12] and RoBERTa [11]. To combine text and author information we also train a multilayer perceptron (MLP) classifier that leverages the BERT-based embedding of the issue with a one-hot encoding representation of the author-issue relation. Both models outperform the baseline and we observe the best performance ($F1 = .8591$) with the model based on textual information only. All the replication material is available on GitHub [13]. We then perform an error analysis which is useful in order to understand the main causes of errors in the classification of the test set.

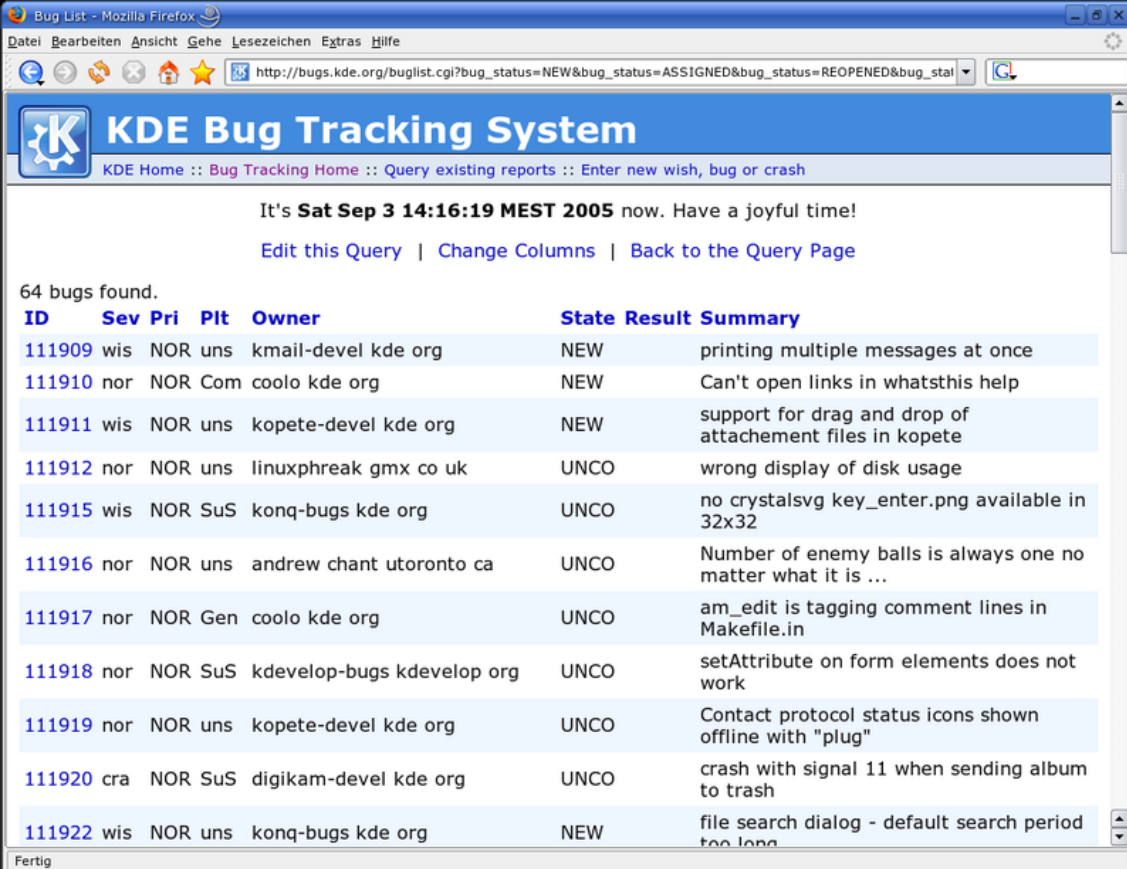
1.1 Thesis structure

In the second chapter, a general theoretical background for issue tracking and pretrained language models is given, along with a glance at the language models that have been taken into consideration for this thesis. In the third chapter we describe the issue report classification task, the NLBSE22 challenge and give all the details about the dataset used for our study. In the fourth chapter we describe the methodology used for solving the issue report classification task. In the fifth chapter we show the results of our studies, followed by a discussion about results and errors performed by the classifier in the sixth chapter. In the seventh chapter we report the conclusions together with possible future works.

2. Background

2.1 Issue Tracking

Just as strong leadership is required to guide a team toward success, so too are strong communication and collaboration tools essential in completing that journey. Much research has been done in the field of computer-supported cooperative work and software engineering to examine how software teams communicate with each other and coordinate their work. The majority of developers spends their time interacting with co-workers [14]. Other studies argued that software development is both knowledge-intensive and collaborative [15]. Development teams use Issue Tracking Systems (ITS) such as Bugzilla, Github Issues, GitLab Issues, JIRA to track issues, including bugs to be fixed or features to be implemented. Over the years ITS have emerged as a central tool for planning and organizing development work [16], and for communicating with users and other stakeholders [17] [18].



ID	Sev	Pri	Pit	Owner	State	Result	Summary
111909	wis	NOR	uns	kmail-devel kde org	NEW		printing multiple messages at once
111910	nor	NOR	Com	coolo kde org	NEW		Can't open links in whatsthis help
111911	wis	NOR	uns	kopete-devel kde org	NEW		support for drag and drop of attachment files in kopete
111912	nor	NOR	uns	linuxphreak gmx co uk	UNCO		wrong display of disk usage
111915	wis	NOR	SuS	konq-bugs kde org	UNCO		no crystalsvg key_enter.png available in 32x32
111916	nor	NOR	uns	andrew chant utoronto ca	UNCO		Number of enemy balls is always one no matter what it is ...
111917	nor	NOR	Gen	coolo kde org	UNCO		am_edit is tagging comment lines in Makefile.in
111918	nor	NOR	SuS	kdevelop-bugs kdevelop org	UNCO		setAttribute on form elements does not work
111919	nor	NOR	uns	kopete-devel kde org	UNCO		Contact protocol status icons shown offline with "plug"
111920	cra	NOR	SuS	digikam-devel kde org	UNCO		crash with signal 11 when sending album to trash
111922	wis	NOR	uns	konq-bugs kde org	NEW		file search dialog - default search period too long

Figure 1. Bugzilla, list of bugs in the KDE project, from [19]

In a project, new requirements are coming constantly. So, it is necessary to have tools which allow somebody to fully and easily:

- share the information across the team;
- have an instant overview of the state of the software;
- expertly decide about releasing;
- set and update the importance of individual fixes and adjustments;
- have a recorded history of changes.

An issue tracking system has the main function to track:

- what should be fixed or created;
- what the bug symptoms and appearances are, what actually doesn't work;
- how it should work the right way;
- who reported the request, who confirmed, analyzed, implemented the solution, and verified it;
- when the request was reported, when it was fixed and when verified;
- what led to the decision to choose one way of fixing instead of another;
- what changes in code were made;
- how long it took to handle the request.

Those insights about the state of the project are really useful for the developing team and, if used in the right way, can bring the following benefits:

- improve the quality of software;
- increase satisfaction of users and customers;
- ensure requests accountability;
- improve communication in the team and also to customers;
- increase of productivity of the team;
- reduce expenses [20].

Issue trackers are often used by open-source development teams and they are a primary and logical location for much of the distributed negotiation involved in resolving bugs, through extended interactions that involve debate among developers, reaching consensus, or soliciting management input [21, 22]. Works on issue tracking systems that has primarily focused on improving the quality of bug reports [16], identifying who should work on a given issue [23], and improving developers' ability to detect defects in their

systems [24] [17].

Issues are units of information usually including a summary (title), a description, and a number of properties like status, priority, and fix version.

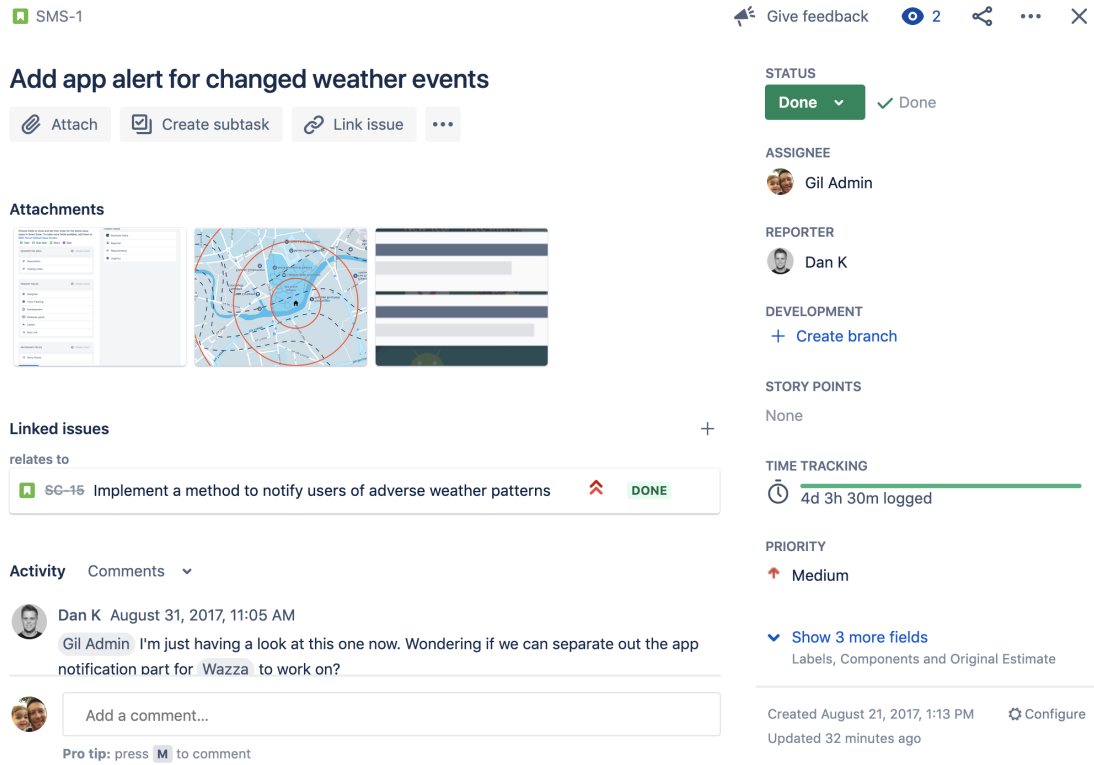


Figure 2. An issue tracked by JIRA, from [25]

A key focus of ITSs is the evolutionary refinement of the issues [26] (also known as iterative improvement), which means that information is gained and refined over time, while developers and stakeholders collaborate to address the issues. Over the last two decades, software engineering research has intensively studied issues and issue trackers, often based on Bugzilla¹, Github², GitLab³ and Jira⁴. The primary research focus has been on the specific issue type of bug reports: the understanding and improvement of information quality therein [16], and the prediction of bug properties such as severity [27, 28], assignee [29], and duplicate reports [30, 31, 32] for supporting software evolution and maintenance. [33]

¹<https://www.bugzilla.org/>

²<https://www.github.com/>

³<https://www.gitlab.com/>

⁴<https://www.atlassian.com/software/jira>

Open Issue #1395 opened about 14 hours ago by **Marcia Ramos** 0 of 2 tasks completed New issue Close issue Edit

GitLab Issue

Hello World! 🙌

This is my issue's description, written in markdown (GitLab Flavored Markdown).

This is an `<h3>`

Let's quote someone here

Add a task list:

- Task 1
- Task 2

Mention merge requests ([gitlab-org/gitlab-ee!1784 \(merged\)](#)) and issues ([gitlab-org/gitlab-ee#2101 \(closed\)](#)) and hover over them to see their titles.

Invite users to collaborate with `@mentions`: @marcia

Edited just now by Marcia Ramos

1 Related Merge Request

🟢 !1784 [How to Configure LDAP with GitLab EE](#) in [GitLab.org / GitLab Enterprise Edition](#) Merged

👍 0 🙌 0 😊 0 Create a merge request

- Marcia Ramos @marcia assigned to @axil and @jivanvl about 14 hours ago
- Marcia Ramos @marcia changed milestone to 9.2 about 14 hours ago
- Marcia Ramos @marcia added `on it` label about 14 hours ago
- Marcia Ramos @marcia changed time estimate to 30m about 14 hours ago

✔ **Create a merge request**
Creates a branch named after this issue and a merge request. The source branch is 'master' by default.

Create a branch
Creates a branch named after this issue. The source branch is 'master' by default.

Write Preview B I 🔗 <> ☰ ☰ ✉ ✕

Write a comment or drag your files here...

Markdown and slash commands are supported 📎 Attach a file

Comment Close issue

Todo Mark done »

3 Assignees Edit

Milestone Edit

9.2

Time tracking 🔔

Estimated: 30m

Due date Edit

May 15, 2017 - remove due date

Labels Edit

`on it`

Weight Edit

3

3 participants

Notifications Unsubscribe

Reference: [gitlab-com/www-g...](#) 📄

Figure 3. An issue tracked by GitLab, from [34]

2.2 Text Classification

Text classification (TC) is the task of assigning predefined categories to free-text documents. It can provide conceptual views of document collections and has important applications in the real world. Instead of manually classifying documents or hand-crafting automatic classification rules, statistical text categorization uses machine learning methods to learn automatic classification rules based on human-labeled training documents [35]. TC is being applied in many contexts, ranging from document indexing based on a controlled vocabulary, to document filtering, automated metadata generation, word sense disambiguation, population of hierarchical catalogues of Web resources, and in general any application requiring document organization or selective and adaptive document dispatching. In the '90s this approach has increasingly lost popularity (especially in the research community) in favour of the machine learning paradigm, according to which a general inductive process automatically builds an automatic text classifier by learning, from a set of preclassified documents, the characteristics of the categories of interest. The advantages of this approach are an accuracy comparable to that achieved by human experts, and a considerable savings in terms of expert manpower, since no intervention from either knowledge engineers or domain experts is needed for the construction of the classifier or for its porting to a different set of categories [36]. It is useful to distinguish among TC problems based on the number of class to which a document can belong:

- Binary classification, if there are only two possible classes (e.g.: spam / non-spam),
- Multi-class classification, if there are more than two possible classes and each document can belong exclusively to one of the classes,
- Multi-label classification, if there are more than two possible classes and each document can belong to two or more classes.

Multi-class and multi-label problems are often faced by reducing the task to k different binary classification subtasks, one for each category.

For each binary classification subtask, the members of the category are treated as positive examples, the others are treated as negative examples [36]. Machine Learning and deep learning based approaches consist in analyzing annotated corpora of texts inferring which features of the text, typically in a bag of words fashion [37] or by n -grams, are relevant for the classification in an automatic way [38]. The most classical approach for text classification consists of extracting basic corpus statistics such as the word frequency or TF-IDF [39] to generate large sparse embedding vectors with a size equal to the

vocabulary size. In these cases, Latent Semantic Analysis [40] may be useful for reducing the dimensionality of such vectors through the Singular Value Decomposition (SVD). As shown in [41], on some occasions, models using TF-IDF, despite being simpler and unable to capture complex text patterns, can achieve better results than more complex neural-based approaches [42]. Neural language models learn to represent textual-tokens (such as words) as dense vectors, referred as to word embeddings, in a self-supervised fashion. These learned representations can then be used for various NLP applications. One popular neural language model is word2vec [43], which learns to map the words that come in similar contexts to similar vector representations. The learned word2vec representations also allow for some simple algebraic operations on word embeddings in vector space [44]. Previous work uses various neural models to learn text representation, including convolution models [45, 41, 46, 47, 48, 49], recurrent models [50, 51, 52], and attention mechanisms [53, 54] [55]. More recently, the Transformers architecture [56], replacing the recurrence with the self-attention mechanism, enabled that large pre-trained language models could now be used to address several NLP tasks, leading to the state-of-the-art in many of these applications [44]. We will go into the details about pre-trained language models in the next section.

2.3 Pretrained Language Models

2.3.1 Transformers and Attention

A **Transformer** is a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output. Most competitive neural sequence transduction models have an encoder-decoder structure. In a transformer, the encoder maps an input sequence of symbol representations (x_1, \dots, x_n) to a sequence of continuous representations $z = (z_1, \dots, z_n)$. Given z , the decoder then generates an output sequence (y_1, \dots, y_m) of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. The transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder.

The *encoder* is composed of a stack of N identical layers. Each layer has two sub-layers. The first is a *multi-head self-attention* mechanism, and the second is a simple, position wise *fully connected feed-forward network*. A *residual connection* around each of the two sub-layers is employed, followed by layer normalization.

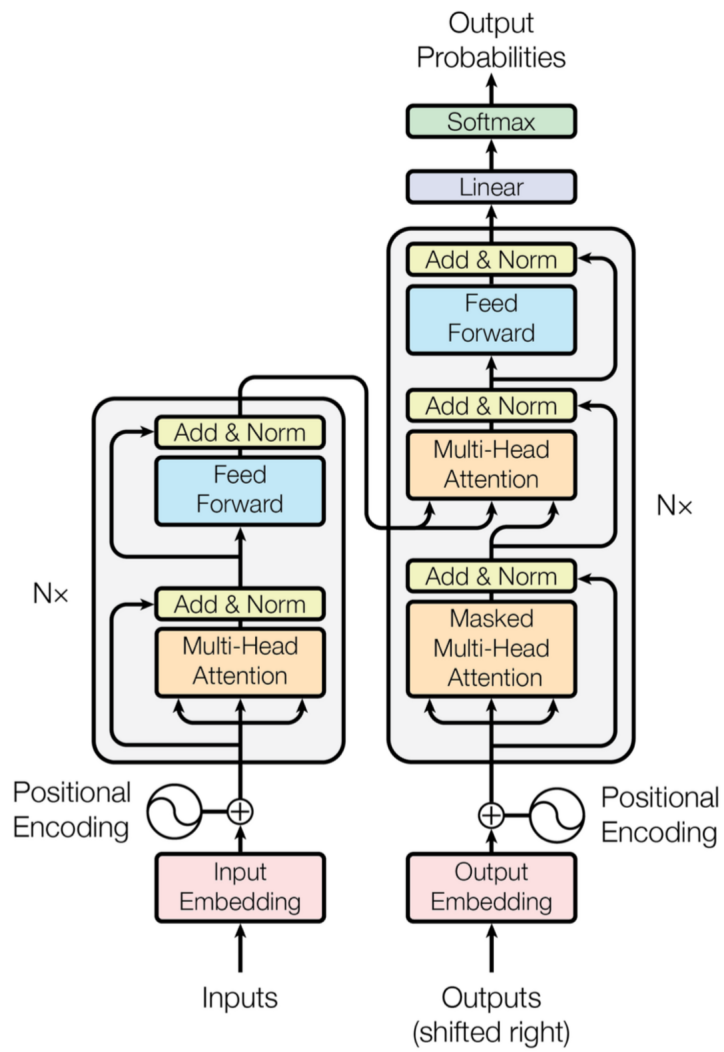


Figure 4. Illustration of Transformers architecture, from [56]

The *decoder* is also composed of a stack of N identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs *multi-head attention* over the output of the encoder stack. Similar to the encoder, residual connections around each of the sub-layers are employed, followed by layer normalization. The self-attention sub-layer is modified in the decoder stack to prevent positions from attending to subsequent positions. This ensures that the predictions for position i can depend only on the known outputs at positions less than i .

The attention function which is used in [56] is called **Scaled Dot-Product Attention**. The input consists of *queries* and *keys* of dimension dk , and values of dimension dv . The dot products of the query with all keys is computed, then divided by \sqrt{dk} , and apply a *softmax* function to obtain the weights on the values. The matrix of outputs is computed as:

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

Instead of performing a single attention function with d_{model} -dimensional keys, values and queries, it is beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k , d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values, the attention function is performed in parallel, yielding d_v -dimensional output values. These are concatenated and once again projected, resulting in the final values. Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O \\ \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned}$$

So we have a new set of parameters (multiple queries, keys and values). The heads are concatenated and multiplied by a new parameter matrix W^O . This setting helps focusing on more parts of discourse. The Transformer uses multi-head attention in three different ways:

- In *encoder-decoder attention* layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence.
- The encoder contains self-attention layers. In a self-attention layer all of the keys,

values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.

- Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. It is important to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections

So, the encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its encoder-decoder attention layer which helps the decoder focus on appropriate places in the input sequence The self attention layers in the decoder operate in a slightly different way, they mask future positions before the softmax step in the self-attention calculation. The encoder-decoder attention” layer works just like multiheaded self-attention, except it creates its queries matrix from the layer below it, and takes the keys and values matrix from the output of the encoder stack. [56, 57, 58]

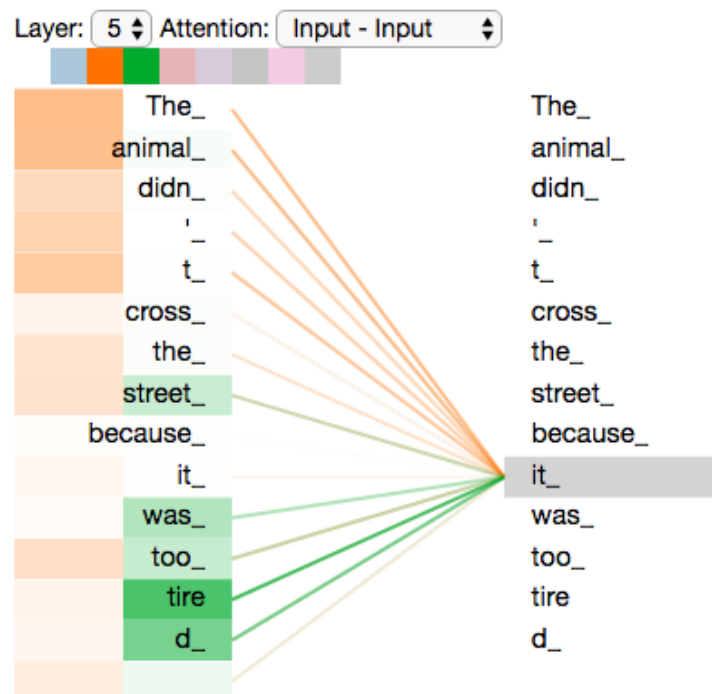


Figure 5. Illustration of the dependencies encoded by the self-attention layers . As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" – in a sense, the model’s representation of the word "it" bakes in some of the representation of both "animal" and "tired", from [57]

2.3.2 BERT

BERT [10] is a language representation model, which stands for "Bidirectional Encoder Representations from Transformers". It is designed to pretrain deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. The pre-trained BERT model can be *finetuned* with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial taskspecific architecture modifications. BERT's model architecture is a multi-layer bidirectional Transformer encoder. There are two existing strategies for applying pre-trained language representations to downstream tasks:

- feature-based,
- fine-tuning

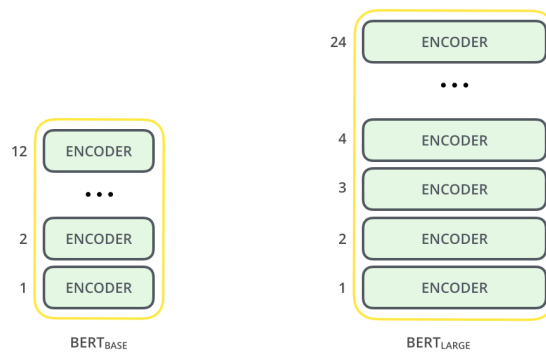


Figure 6. Different variants of BERT, from [59].

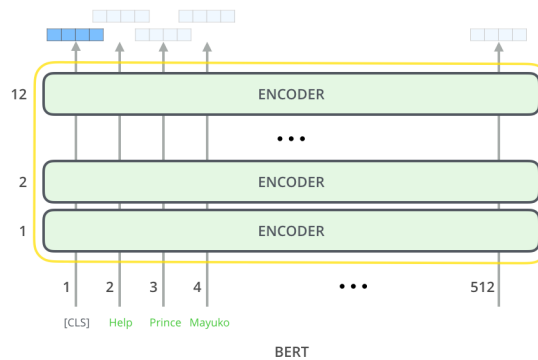


Figure 7. BERT creating word embeddings, from [59].

Feature-based approach, such as ELMo [60], uses task-specific architectures that include the pre-trained representations as additional features.

Fine-tuning introduces minimal task-specific parameters, and is trained on the downstream

tasks by simply fine-tuning all pretrained parameters.

The major limitation is that standard language models are unidirectional, and this limits the choice of architectures that can be used during pre-training. BERT alleviates the previously mentioned unidirectionality constraint by using a *masked language model* (MLM) pre-training objective. The masked language model randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked word based only on its context. The MLM objective enables the representation to fuse the left and the right context, which allows us to pretrain a deep bidirectional Transformer. In addition to the masked language model, the *next sentence prediction* task is used, that jointly pretrains text-pair representations.

After BERT's release, Facebook researchers found out that it was undertrained and that the next sentence prediction task was not so crucial in the training process. RoBERTa [11] has the same architecture as BERT but is trained on a bigger dataset, with longer sequences, without using the NSP task and with some small changes to the masking process used in the MLM task. AIBERT [12] is a BERT model which leverages parameter reduction techniques to lower memory consumption and increase the training speed of BERT.

3. Issue Report Classification

3.1 Related work

Issue tracking systems are important means for maintainers to enable rigorous yet effective software evolution tasks. In issue tracking systems maintainers report tickets or potential problems, manage them and keep track of their progress. But as useful issue tracking systems might be, many developers still end up with a rapidly growing workload and lose control of it [61, 62]. Github¹ provides an integrated lightweight issue tracking system, in which issue submitters are only required to provide a short textual abstract, containing a title and an optional description to report a new issue to a project hosted on GitHub. While this simplified process of reporting issues decreases the barrier to entry and attracts more inexperienced external contributors, it complicates the work of the development teams for maintaining the software, as several hundreds of issues of different nature (e.g., asking questions, proposing features, signaling bugs) and quality are usually submitted [63].

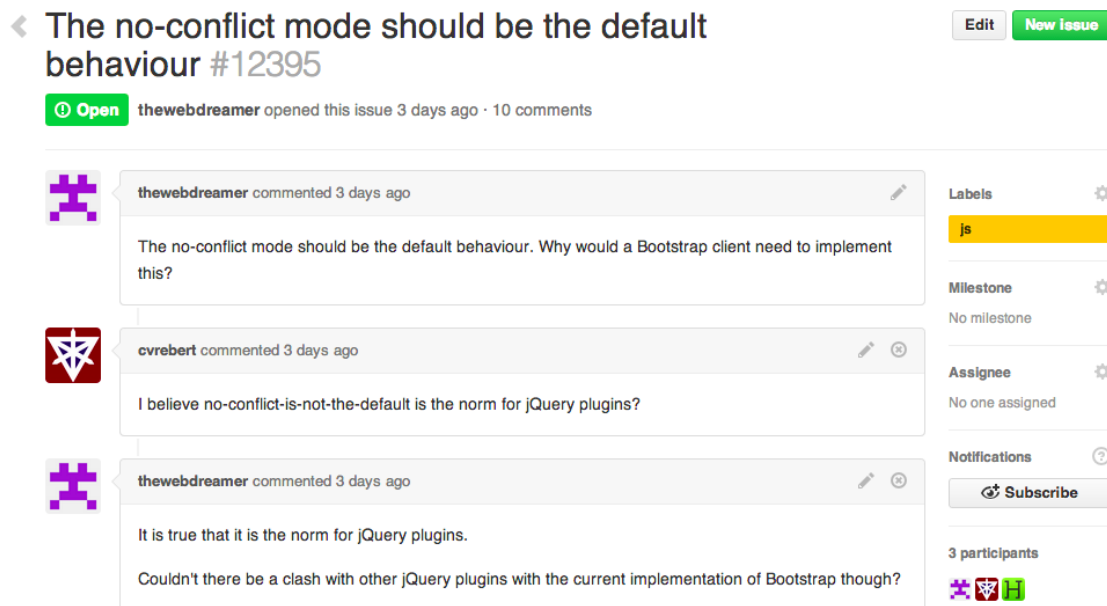


Figure 8. Example of GitHub issue, from [64].

To cope with these problems, GitHub also offers a customizable labeling system, which can be used by developers to mark and manage issue reports. In particular, labels can

¹<https://www.github.com/>

give immediate clues about the issues (e.g., what sort of topic the issue is about, what development task the issue is related to, or what priority the issue has) and are also useful for project administrators, since they can serve both as classification and filtering mechanism, thus facilitating the managing of the project [65]. However, manually assigning labels to issues is a labor-intensive and time-consuming task for project managers [63]. Indeed, although labeling has a positive impact on the effectiveness of issue processing [66], the labeling mechanism is scarcely used on GitHub [61] [7].

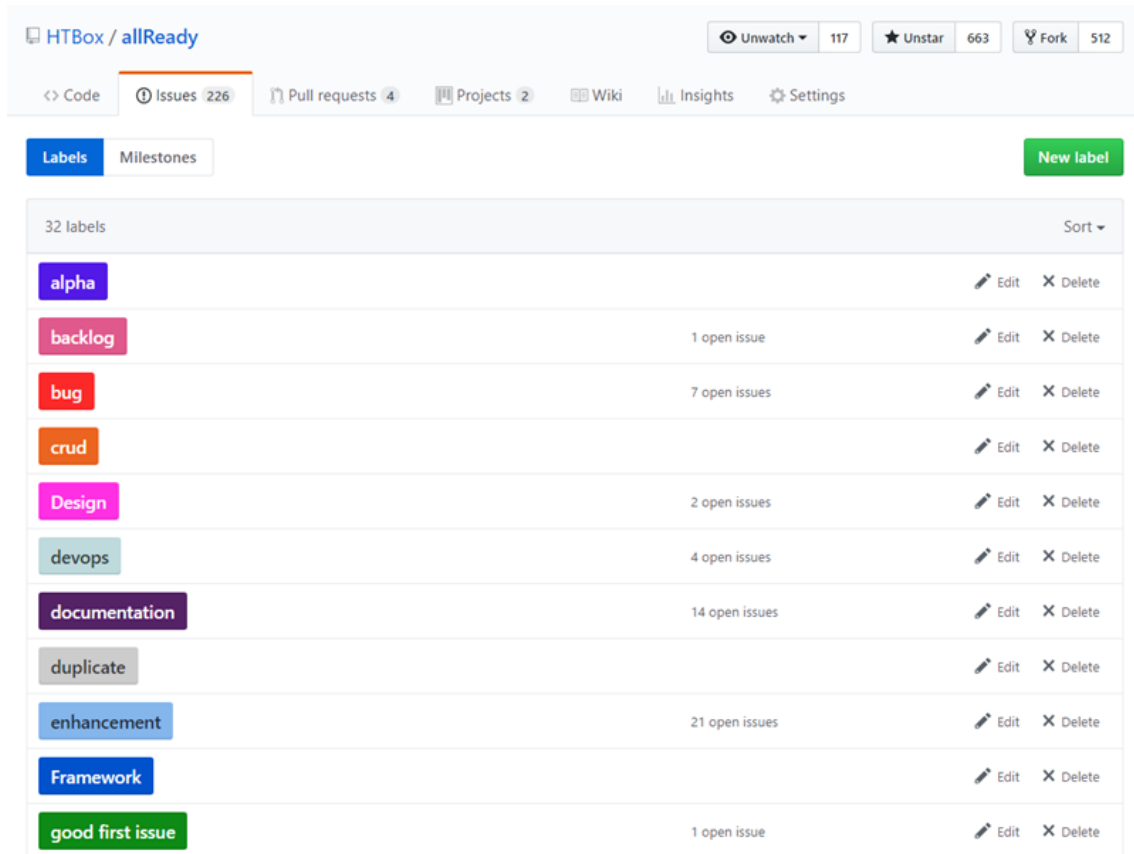


Figure 9. Custom labels in GitHub Issues, from [67].

Previous studies presented several approaches to automatically categorize issues posted in bug tracking systems. For example, it is shown that machine learning models can be used in order to discriminate bugs from other kinds of issues. In [2], six different issue categories are introduced – bug, feature request, improvement request, documentation request, refactoring request, and others – and demonstrated that often developers and maintainers assign the wrong issue category to the reports. To address this problem, in [68] structured data with unstructured free-text data are combined to train a classifier able to predict with high accuracy if a bug report is actually a bug or another kind of issue. Unfortunately, no structured information could be found on GitHub issues, according to the GitHub issue tracking lightweight structure [7]. Recently, Kallis et al. [7] proposed

Ticket Tagger, a machine learning classifier that predicts the label to assign to issues trained on GitHub data. Specifically, Ticket Tagger leverages only the textual content of an issue title and body, whose vectorial representation is based on *fastText* [8], an open-source tool released by Facebook AI research.

3.2 Challenge description

In this work we describe the systems we developed to participate in the tool competition of NLBSE'22 on automatic labeling of GitHub issues. NLBSE'22 is the 1st International Workshop on Natural Language-based Software Engineering. It was co-located with ICSE 2022 and was held on the 8th of May 2022. The first edition of the NLBSE'22 tool competition was on automatic issue report classification, an important task in issue management and prioritization.

NL-BASED SOFTWARE ENGINEERING

ORGANIZATION TOOL COMPETITION KEYNOTE & TUTORIAL PROGRAM VENUE

NLBSE 2022

The 1st Intl. Workshop on Natural Language-based Software Engineering
Co-located with ICSE 2022
May 8, 2022

Tool Competition

Tool competition slides

Introduction

NLP-based approaches and tools have been proposed to improve the efficiency of software engineers, processes, and products, by automatically processing natural language artifacts (issues, emails, commits, etc.).

We believe that the availability of accurate tools is becoming increasingly necessary to improve Software Engineering (SE) processes. One important process is issue management and prioritization where developers have to understand, classify, prioritize, assign, etc. incoming issues reported by end-users and developers.

Important Dates

- PAPER/TOOL SUBMISSION**
February 21, 2022
- ACCEPTANCE AND COMPETITION RESULTS NOTIFICATION**
March 4, 2022
- CAMERA-READY PAPER SUBMISSION**
March 22, 2022

All dates are Anywhere on Earth (AoE).

Figure 10. Tool Competition page of the NLBSE'22 workshop

For the competition, a dataset encompassing more than 800k labeled issue reports (as bugs, enhancements, and questions) extracted from real open-source projects was provided. The goal was to develop a classification model leveraging this dataset and compare the achieved results against a proposed baseline approach, TicketTagger (based on FastText) [7, 6]. The submissions were ranked based on the F1-micro score achieved by the proposed classifiers on the test set, as indicated in the papers. While the F1-score was used for ranking the

models and determining the winner of the competition, the participants were also asked to report the following metrics: precision, recall and F1 for each class. We report the formulas below. TP, FP, TN, and FN indicates number of True Positives, False Positives, True Negatives, and False negatives, respectively. We denote a generic class with c .

$$P_c = \frac{TP_c}{TP_c + FP_c}$$

$$R_c = \frac{TP_c}{TP_c + FN_c}$$

$$F_{1,c} = \frac{2 \times P_c \times R_c}{P_c + R_c}$$

$$P = \frac{\sum_c TP_c}{\sum_c (TP_c + FP_c)}$$

$$R = \frac{\sum_c TP_c}{\sum_c (TP_c + FN_c)}$$

$$F_1 = \frac{2 \times P \times R}{P + R}$$

Note that micro-average precision and recall are the same as micro-average F1-score. Participants were free to select and transform variables from the training set, but no new sources could have been added. In other words, any inputs or features used to create the classifier, had to be derived from the provided training set. Practises like preprocessing, sampling, over/under-sampling, selecting a subset of the attributes, perform feature-engineering, split the training set into a model-finetuning validation set, etc. were allowed. The evaluation was performed on the entire test set only. No sampling, rebalancing, undersampling or oversampling techniques were allowed on the test set [69].

3.2.1 Dataset

The issues in the dataset were extracted from The GitHub Archive [70] using Google BigQuery [71]. The dataset consists of more than 800K Github issue-reports extracted from real open-source projects. The organizers of the tool competition selected all the closed issues during the first semester of 2021 (from January 1st 2021 to May 31st 2021)

that contained any of the labels bug, enhancement, and question at the issue closing time. The dataset was given in CSV format without applying any preprocessing on the issues [69]. We use this dataset, distributed by the tool competition organizers [9, 6, 7], to take part to the NLBSE'22 Tool Competition.

For each issue, the following features were collected and made available as dataset:

- issue url
- label
- creation date
- issue author association
- repository url
- title
- body

The label can be one of the following:

- bug, which means that the issue contains a bug report to be fixed,
- enhancement, issues which contain improvement and new feature requests,
- question, asked by an user about the usage of the software.

Labels can be assigned by the user who opened the issue or by repository maintainers. In case of multiple labels, the most recent is taken as ground truth. The issue author association is the role played in the repository by the person who opened the issue. It can have the following values:

- owner,
- contributor,
- member,
- collaborator,
- none,
- mannequin.

From the set of features, the more relevant for our task are title, body, issue author association and obviously, the label. The text of title and body is written in Markdown format. The tool competition organizers distributed the dataset already split in train and test set, as shown in Table 1. Both the training and the test set were available at the beginning

of the Tool Competition.

Table 1. Dataset with label distribution.

	Train set		Test set	
bug	361,103	(50%)	40,288	(50%)
enhancement	299,374	(41%)	33,203	(41%)
question	62,422	(9%)	7,027	(9%)
total	722,899		80,518	

The distribution of the labels is the same for both training and test set. Labels are unbalanced, with bugs (50%) and enhancement (41%) being better represented than question (9%), which is the minority class.

4. Methodology

4.1 Research questions

With our work we aim at understanding if the issue report classification task can be faced using only the textual information, using state of the art pretrained language models.

RQ1: To what extent we can leverage pre-trained language models to build an automatic classifier for GitHub issue labeling?

Furthermore, we aim at understanding if including non-textual information about the issues, such as the issue-author association can improve the performance.

RQ2: To what extent the issue-author association contribute to improve the performance of a classifier modeling textual information based on pre-trained models?

Unfortunately, other non-textual features were not available and it was not possible to integrate them, as specified in the competition rules. During the competition we analysed manually some of the issues, noticing that data were very noisy. So, as a follow-up study after the competition deadline, we tried to filter out that noise from our data to achieve better performances.

RQ3: How can we filter out noisy data? And how will the classifier perform on the filtered data?

4.2 Pre-processing

As a first pre-processing step, we identify text patterns indicating non-textual items such as images, links, code snippets and replace them with ad-hoc tokens (e.g `` for images). Then, we perform further text normalization step using *ekphrasis Text Pre-Processor*¹, which is able to identify other patterns such as:

- URLs,

¹<https://github.com/cbaziotis/ekphrasis>

- email addresses,
- percent or currency symbols,
- phone numbers,
- user mention,
- time,
- date,
- numbers.

This preprocessing step is very common in other studies using pretrained language models [72, 73, 74, 75, 76]. We replace those items with *ad hoc* tokens. We use *ekphrasis* also to unpack hashtags, contractions and emojis. The idea is that we don't want to model the link string, the image name, or the code in our text. We just want to model the presence or not of such elements. We want to avoid modeling code as normal text in order to avoid bias on some specific repositories or specific code snippets. Our model should be able to classify new issues incorporating unseen code (maybe using brand new repositories, which are never used in the training set). Since the documents will be fed into either BERT or its variants, we encode all the documents in the dataset using the model-specific tokenizer. To avoid exceeding the GPU memory capacity, we pad/truncate each document to 128 tokens, in line with previous work [4].

4.3 Model fine-tuning

We implement a supervised approach by leveraging state-of-the-art models based on transformers. Specifically, we experimented with fine-tuning of BERT-based models in two different settings as depicted in Figure 11:

- In the first setting (Classifier 1 in the figure), we leverage the text content of the issue (title and body) and fine-tune the language model to obtain the final classifier.
- In the second setting (Classifier 2 in the figure), we combine the textual information with the information provided by the author-association field and train a feed-forward network.

As a preliminary step to both approaches, we need to identify the best pre-trained language model to use for the issue classification task. In this study, we experiment with three state of the art pretrained language models:

- BERT [10],

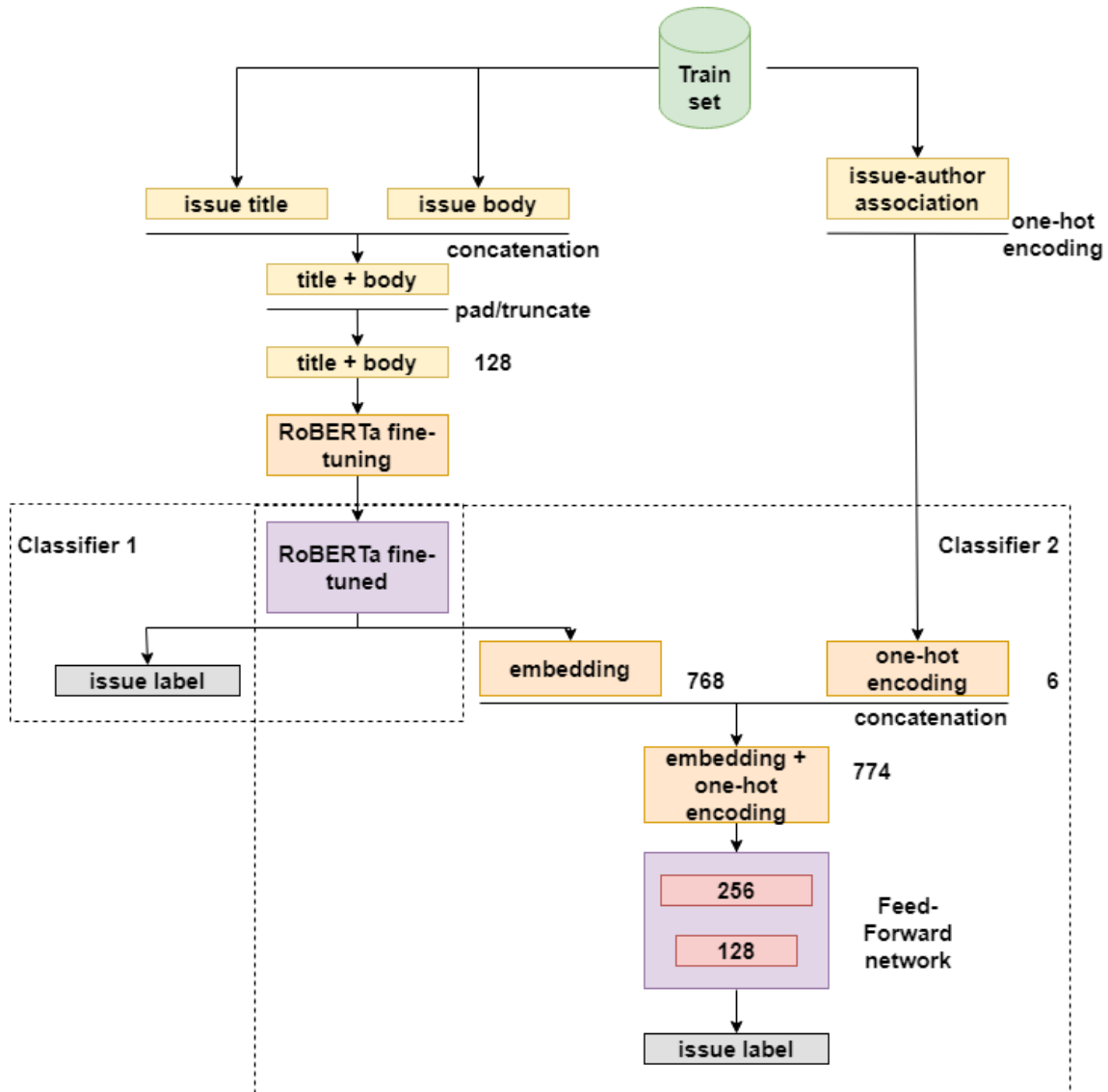


Figure 11. The two classifiers implemented for issue labeling.

- ALBERT [12],
- RoBERTa [11].

For BERT, we use both the base model and the large model. To select the best model, we perform fine-tuning of each model using the train set. Specifically:

- We split the train set in two subsets, one containing 90% of the issues and the other containing 10% of the issues.
- We train each of the models on the first subset.
- We test each of the models on the second subset (validation set)
- We compare the performances of all the models to select the best one.

This procedure is illustrated in Figure 12. For each language model, the fine-tuning phase

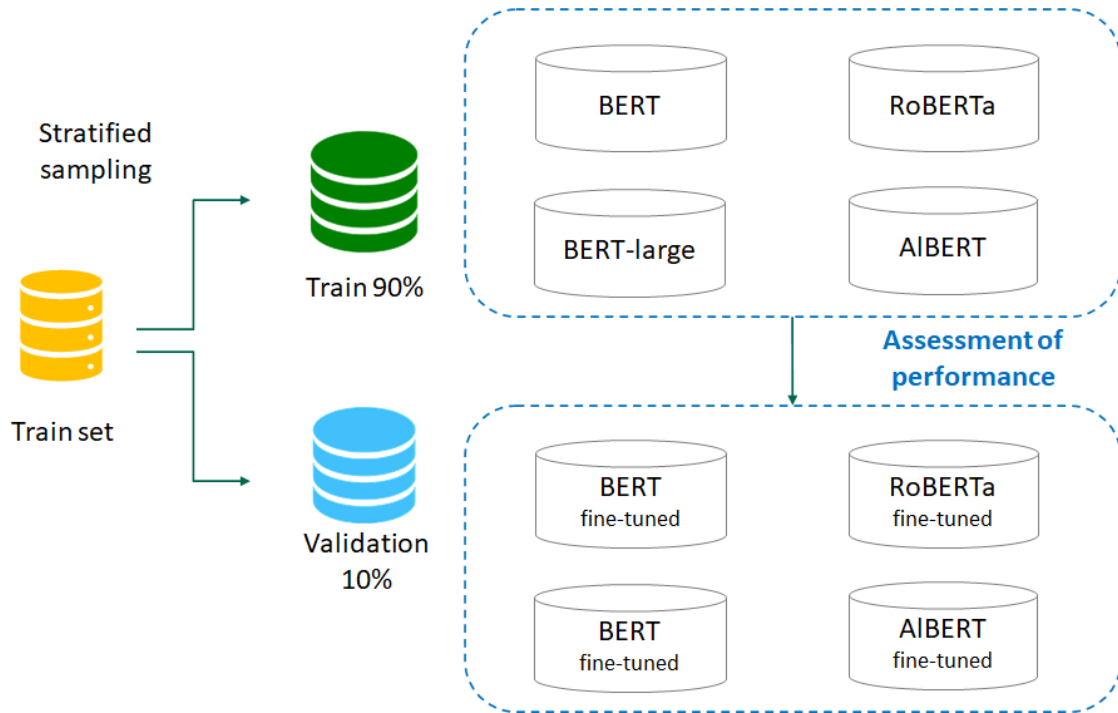


Figure 12. Illustration of the process adopted to select the best BERT model

is done in 4 epochs. To select the best number of epochs for each model, we test all the models after each epoch. In the training phase we used the Adam optimizer with weight decay. As optimizer parameters we used a learning rate = 2×10^{-5} and a epsilon = 1×10^{-8} .

Table 2 reports the results of the performance assessment on the validation set for all models we experimented with. Given the small differences in the overall micro average F1 observed for all models, we decided to pick as best model the one achieving the best F1 on the minority class, which is the *question* class. As a result of this validation phase, we select RoBERTa as the most promising language model to be used for training the classifier for the challenge submission.

4.4 Training the Issue Classifiers

Based on the performance observed on the validation set, we decided to use RoBERTa for training the classifier for the challenge submission, using the full train set provided by the organizers. As a first step, we fine-tune RoBERTa using the full train set provided by the organizer. We replicate the same procedure adopted for model selection, i.e. we fine-tune RoBERTa using the issue title and body, which we pad/truncate to consistently

Table 2. Model selection: the best performance achieved on the validation set for all fine-tuned models.

	ALBERT (3 epochs)			BERT-base (2 epochs)			BERT-large (2 epochs)			RoBERTa (4 epochs)		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
bug	.8695	.8906	.8799	.8712	.9069	.8887	.8694	.9106	.8895	.8756	.8985	.8869
enhancement	.8615	.874	.8677	.8709	.8802	.8756	.8722	.8763	.8742	.8743	.8755	.8749
question	.6734	.532	.5944	.7142	.5083	.5939	.7257	.5104	.5993	.6667	.5612	.6094
micro avg	.8528	.8528	.8528	.8614	.8614	.8614	.8618	.8618	.8618	.8599	.8599	.8599
macro avg	.8015	.7655	.7807	.8188	.7651	.7860	.8224	.7658	.7877	.8055	.7784	.7904

represent documents with the same length (128 tokens). Then, we use the fine-tuned RoBERTa model for building the two classifiers. For Classifier 1, we simply rely on the textual information of the GitHub issues, that is on the concatenation of each issue title and body. For Classifier 2, we build a multilayer perceptron (MLP) classifier that leverages the combination of the textual information of the issues with the information regarding the issue-author association contained in the dataset:

- We extract the RoBERTa-based embeddings of each document, i.e., the concatenation of the title and body of the issues, using the last hidden layer before the classification layer of the fine-tuned model, obtaining a 768 dimension embedding.
- We compute the one-hot encoding vectors for each value of the issue-author association attribute (six dimensions overall, one for each possible value of the issue-author association attribute).
- We then concatenate the RoBERTa-based embedding with the one-hot-encoding representation of the issue-author association information, as illustrated in 13.

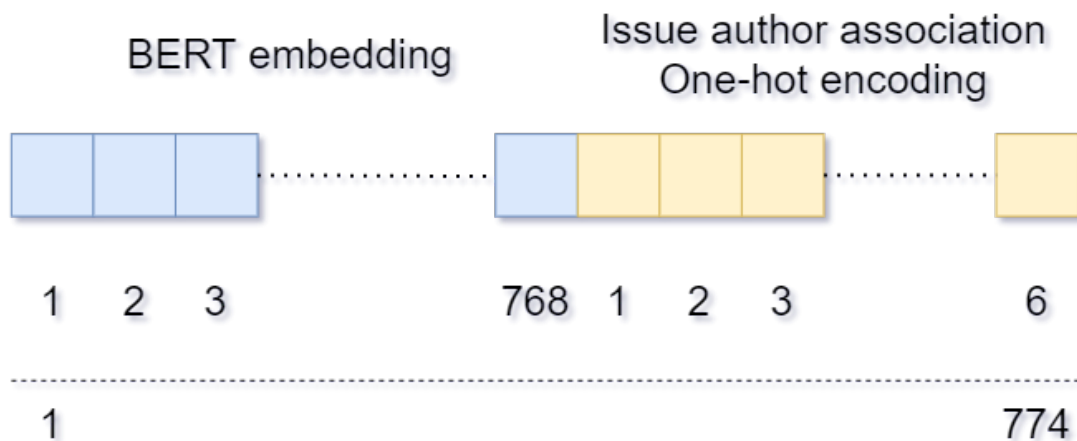


Figure 13. Concatenation of the BERT embedding with the issue author association one-hot encoding vector

The new vector is fed into a multi-layer perceptron with two hidden layers of size 256 and 128, respectively. In order to train the network, we use stratified sampling to split the training set into train (90%) and a validation set (10%). The network is then trained with the following parameters:

- batch size = 32,
- learning rate = 1×10^{-5}
- Adam optimizer with learning rate = 2×10^{-5} and a epsilon = 1×10^{-8} .

- epochs = 100

We set up and use an early stopping criterion with patience = 5. We use a callback function to select the model achieving the best performance once the early stopping condition is verified. For the training, we use *NLLLoss* and set the weights of the loss function as inversely proportional to the class frequencies in the training data.

4.5 Evaluation

In line with the guidelines of the challenge, we provide the evaluation of the two classifiers on the test set in terms of micro-F1. Given the unbalance distribution of the labels in the dataset, we also report the macro-F1 because micro-averaging is known to be influenced by the performance on the majority class. Conversely, the ability of a classifier to correctly identify items belonging to classes with few training instances is correctly assessed by the macro-average. To address the problem of class imbalance in the training data, we also experimented with undersampling, thus obtaining a balanced dataset based on the number of items included in the minority class of *questions*. However, we observed a worse performance with respect to the one observed when the full train set in use. As such, we report the performance of the models trained using the complete train set. The goal of our experiments is twofold. On the one hand we compare our approaches with the performance of Ticker Tagger, the FastText-based approach provided as a baseline by the challenge organizer. Ticket Tagger was originally trained and validated on 30,000 GitHub issues [6, 7]. To compute the baseline performance for the tool competition, its performance was reassessed on the challenge test set using the Colab notebook provided by the organizers. We report the baseline performance in Table 3. We also compare our models with the other ones submitted for the tool competition. On the other hand we aim at assessing to what extent a simple approach based on textual information enable automatic labeling of GitHub issues.

5. Results

In Table 3, we report the performance of the two classifiers and provide comparison with the baseline approach based on *fastText*. Both our classifiers outperform the baseline and they achieve a performance comparable to the one reported by previous work on issue classification based on contextual embeddings [5], as done in this study. In particular, Classifier 1 (RoBERTa fine-tuned) achieves the best micro F1 (.8591). As for Classifier 2 (MLP), which also includes consideration of the author-issue association, we observe a lower micro F1 (.8295). However, the recall for the minority class *question* is substantially improved up to .7537, as also reflected by the higher macro average recall (.7774 and .8092 for Classifier 1 and 2, respectively). Albeit the overall performance is substantially unvaried in terms of micro F1, the choice between the RoBERTa-based and MLP-based for practical usage might not be trivial as RoBERTa optimizes the precision of the minority class while the MLP achieves a better recall.

For the sake of the challenge submission, we identify the RoBERTa-based classifier as the best performing one, given its higher micro-average F1.

Table 3. Performance of the system on the test.

Class	Classifier 1: RoBERTa <i>Title + Body</i>			Classifier 2: MLP <i>Author + Title + Body</i>			FastText Baseline <i>Title + Body</i>		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
bug	.8750	.8988	.8867	.8934	.8346	.8630	.8314	.8725	.8515
enhanc.	.8713	.8743	.8728	.8797	.8394	.8591	.8155	.8464	.8307
question	.6760	.5591	.6120	.4727	.7537	.5810	.6521	.3502	.4557
micro avg	.8591	.8591	.8591	.8295	.8295	.8295	.8162	.8162	.8162
macro avg	.8074	.7774	.7905	.7486	.8092	.7677	.7663	.6897	.7126

5.1 Comparison

Here we report a general overview of the tool competition submissions and results.

- Izadi [77] proposed CatIss, a fine-tuned pretrained RoBERTa model [11] that uses (as input) the issue text (title and body) concatenated with the issue timestamp, author, and repository (the owner and repository name). The processing of the issues included removal of exact duplicate issues (performed on the training set only), text

normalization to replace content with a predefined tag (e.g., <FUNCTION> for function names), special character removal, and lower-casing. Izadi also reported a Logistic Regression model as an additional baseline model [69].

- Bharadwaj and Kadam [78] proposed multiple classifiers based on BERT (vanilla BERT [10], CodeBERT [79], and RoBERTa [11]) and XLNet [80] to encode the issue text (title and body) as embeddings. These embeddings are combined with embeddings obtained from additional issue features, namely whether or not the issue was submitted early in the project history (defined based on a threshold), the project owner, and whether or not the issue title describes a question. The combined embeddings are the input to classification layers. The BERT-based models used by the authors were also fine-tuned. The main preprocessing applied include regex-based substitution of code snippets, URLs, usernames, and numbers with predefined tags [69].
- Siddiq and Santos [81] proposed a BERT-based classifier, finetuned using the issue title and body. Preprocessing included removal of repeating white space characters and replacement of tabs and line breaks with spaces [69].
- Trautsch and Herbold [82] fine-tuned seBERT [83], a model for the software engineering domain that is pretrained using posts from Stack Overflow and issues/commit messages from the repositories of open source projects. The model was fine-tuned using the issue text (title and body) after preprocessing (e.g., replacement of line breaks with spaces and removal of repeating white space characters) [69].

In Table 4 we report the NLBSE'22 Tool Competition Ranking, from [69].

Based on the classification results, the final rank was the following:

- Izadi [77] took the first place in the competition with their CatIss approach (0.872 micro avg. F1-score) [69].
- Bharadwaj and Kadam [78] occupied the second place with their RoBERTa and CodeBERT[79] approach (0.865 and 0.862 micro avg. F1-score, respectively) [69].
- The remaining teams (Colavito et al. [84], Siddiq and Santos [81], and Trautsch and Herbold [82]) finished in the third place of the competition as their best classifiers achieved virtually the same performance (0.855 - 0.859 micro avg. F1-score) [69].

The main difference from our approach and the top performing ones are the following:

- Both Izadi [77] and Bharadwaj & Kadam [78] use a bigger input size for their

Table 4. Issue classification results for bugs, enhancements and questions. The models are ranked by Avg., the micro average precision/recall/F1-score over the three issue types.

Classification Model	Metric	Bug	Enh.	Que.	Avg.
CatIss (RoBERTa) by Izadi[77]	Precision	0.894	0.874	0.720	0.872
	Recall	0.897	0.885	0.664	
	F1-score	0.896	0.879	0.691	
RoBERTa by Bharadwaj & Kadam[78]	Precision	0.872	0.879	0.714	0.865
	Recall	0.911	0.877	0.539	
	F1-score	0.891	0.878	0.614	
CodeBERT by Bharadwaj & Kadam[78]	Precision	0.883	0.866	0.693	0.862
	Recall	0.894	0.891	0.551	
	F1-score	0.888	0.878	0.614	
RoBERTa by Colavito et al.[84]	Precision	0.875	0.871	0.767	0.859
	Recall	0.898	0.874	0.559	
	F1-score	0.886	0.872	0.612	
BERT by Siddiq & Santos[81]	Precision	0.883	0.859	0.678	0.858
	Recall	0.888	0.888	0.546	
	F1-score	0.885	0.873	0.605	
seBERT (BERT) by Trautsch & Herbold[82]	Precision	0.866	0.864	0.731	0.857
	Recall	0.906	0.877	0.487	
	F1-score	0.886	0.871	0.584	
XLNet by Bharadwaj & Kadam[78]	Precision	0.879	0.853	0.706	0.856
	Recall	0.885	0.890	0.534	
	F1-score	0.882	0.871	0.608	
BERT by Bharadwaj & Kadam[78]	Precision	0.875	0.866	0.660	0.855
	Recall	0.892	0.871	0.570	
	F1-score	0.883	0.868	0.611	
MLP by Colavito et al.[84]	Precision	0.893	0.879	0.472	0.829
	Recall	0.834	0.839	0.753	
	F1-score	0.863	0.859	0.581	
Logistic Regression by Izadi[77]	Precision	0.841	0.822	0.655	0.822
	Recall	0.867	0.850	0.432	
	F1-score	0.854	0.835	0.521	
Baseline (fastText) by Kallis et al.[6, 7]	Precision	0.811	0.844	0.669	0.818
	Recall	0.904	0.815	0.336	
	F1-score	0.855	0.830	0.447	

models, the first 200, the second 512. This slows down by a lot the training phase, and doesn't lead to significative improvements.

- Izadi [77] uses as part of the input of their model two more features:
 - repository name,
 - issue creation timestamp.

We believe that those two features can be a source of bias and label leakage. The model learns the names of the repository in the training set and can potentially map timestamps to labels. Still this suggests that different project may have different labelling rationale and then, knowing the name, we can make a distinction between projects and behave differently in the classification.

- Izadi [77] removes duplicates from the training set.

In general, the performances among the participants of the tool competition are very similar, and every team used a pre-trained language model. Also, there are three entries using RoBERTa, which occupies 3/4 of the top four ranking models. All the BERT base models overcome the fastText baseline [6, 7] by 0.04 – 0.06 in the overall F1 micro. Anyways, fastText is way faster, both in training and inference time than a BERT model [85]. So, a non-trivial question is: Is it worth that low increase in performance? A BERT model requires more memory, is slower and heavier than fastText. So, the fastText model is probably easier to deploy and can be used with less resources than a BERT model.

6. Discussion

6.1 Error Analysis

The goal of this analysis is to understand which are the main causes of the errors in the classification of test set’s issues. We formulate some hypothesis:

- The **model** might be not appropriate:
 - Need to find better hyperparameters for the selected model
 - Need to explore other models which might be more useful for the task
- The available **data** might not be enough for classifying correctly the issues:
 - Need to integrate more features which might help in the prediction
 - Select an appropriate way to embed all the knowledge together, textual or not
 - Need more examples of issues labeled as question in order to learn a better model with a more balanced dataset
- The **labels** might be wrong:
 - We cannot trust a random user labelling an issue
 - We cannot trust the labelling criteria of small projects, that can be personal projects or not projects at all [86].
- Every project has a **labelling criteria** and they might not agree.

From the comparison of the tools submitted for the challenge, BERT models are the top performing models. In fact, all the models achieve very similar performances. While tools using other standard machine learning approaches are down in the ranking. So, actually BERT models are the best choice for facing the task.

In Table 5, we report the confusion matrix for our RoBERTa classifier. We observe that the misclassification of *questions* as *bugs* is main cause of error (27% of test documents), immediately followed by misclassification of *questions* as *enhancements* (17% of cases). As the third most frequent cause of error, we observe misclassification of *enhancements* as *bugs* (10%). We conjecture this can be explained by the unbalanced distribution of labels in the dataset (see Table 1). The classifier hasn’t seen a lot of question example, and because of this, can have difficulties in model that kind of issues. A more balanced dataset might help in having more acceptable performances for the question class. To get

Table 5. Confusion matrix on the test set for Classifier 1

<i>Gold label</i>	<i>Classifier prediction</i>		
	bug	enhancement	question
bug	36,210 (90%)	3,106 (8%)	972 (2%)
enhancement	3,261 (10%)	29,031 (87%)	911 (3%)
question	1,914 (27%)	1,184 (17%)	3,929 (56%)

a deeper insight on the difficulties inherent in issue classification, we perform an error analysis by manually inspecting the classification output of the RoBERTa fine-tuned model. Specifically, we examined 370+ cases, that represents a statistically significant sample (with 95% confidence level) of cases for which the classifier yielded a wrong prediction. We observed that some issues labeled as *question* actually report inconsistent behavior or missing code, thus resembling the structure and content of *bug* reports (e.g., "*Fragrance not showing in Homekit - I cannot see the installed fragrance in HomeKit, however it is available in Homebridge.*"). In this cases, it is impossible to distinguish among bugs and question. A bug is a bug if the behaviour of the code is incorrect, and the error does not depend on the user. On the contrary, a question about an error, stands as a question if the behaviour of the code is correct, and the error is caused by the user. Questions often also contain an error message, as also common for bugs. These cases are labeled as question in line with the information seeking goal of the author. However, a text-based classifier might not be necessarily able to disambiguate between bugs and questions in similar cases. A similar situation is observed for *questions* or *bugs* that also include suggestions for fixing the reported problem, which is probably the cause for misclassification as *enhancement*. Finally, the dataset contains issues collected from different projects, thus reflecting possible inconsistencies in the labeling rationale, as well as a few cases not in English. Since our model is pre-trained on an English corpus, it cannot understand other languages and might make errors classifying issues written in languages which he has never seen on the training corpus.

6.1.1 Error Examples

We report here some significative examples of issues which are hard to classify correctly. The issues are taken from the ones misclassified by our model.

In Figures 14 and 15 there is an issue labeled as question. But the content of the issue is a bug report. At write time, the creator of the issue does not assign any issue label. The issue contains a bug report and some instructions on how to reproduce the error. Then, in

Figure 15 a repository member starts handling the issue, and after a day, answers to the issue saying: "This is the expected behaviour of the code". Meaning that the code, used that way, should throw an error. After the analysis, the repository member labels the issue as a question.

Logout handler does not call the auth0 logout endpoint if the user is unauthorized #362

Closed Romainpaulus opened this issue on 8 Apr 2021 · 7 comments

Romainpaulus commented on 8 Apr 2021 · edited

Description

When a user enters a valid username/password through on the universal login sign-in page, but when an auth0 rule returns an `UnauthorizedError` (like when forcing email verification), the user is stuck in an unauthorized state, and redirecting them to `/api/auth/logout` or `/api/auth/login` doesn't let them sign out and try again. The most likely reason is because `sessionCache.isAuthenticated` fails in the logout handler, so the logout handler never executes `client.endSessionUrl`, keeping the SSO cookie of that user on the auth0 domain.

Reproduction

App configuration:

- Create a new auth0 app of type "Regular web application" and using the "Classic" universal login
- Go to [login page customization](#), turn on "Customize login page", add the `loginAfterSignUp: false` option in `var lock = new Auth0Lock(...)`, and save it
- Create a [new auth0 rule](#)
- Select the "Force email verification" rule, save it as is, and make sure the rule is activated
- On the nextJS app, use all the default handlers in `/api/auth/[...auth0].js`

User actions:

- Go to `/api/auth/login`
- Sign up a new user with a valid email/password

Assignees: No one assigned

Labels: **question**

Projects: None yet

Milestone: No milestone

Development: No branches or pull requests

Notifications: Subscribe
You're not receiving notifications f

5 participants

Figure 14. First case: an issue labeled as question, which actually contains a bug report

But what would have happened if that error was not the expected behaviour of the code? Probably the issue text would have been the same, but the member of the repository, knowing how the code should behave in certain situations, would have labeled it as a bug. So, what's the difference among a question and a bug? Is it the behaviour of the code? If it is, then we cannot expect to be able to distinguish among bugs and questions without knowing the expected behaviour of the code. In the issue text there is not such information.

In Figure 16 there is an issue with two labels. At write time the issue creator, which is also a contributor for the repository, labels the issue as a question. In fact, the text is actually a question on repository usage. The issue creator wants to know how to do a specific thing with the repository. Then, a repository collaborator handles the issue, writing: "This isn't possible at the moment" and shows interest in integrating that feature in the repository. After that, he labels the issue as an enhancement. And, in the competition dataset, this issue is labeled as an enhancement.

- You should see a message about a verification email being sent, *do not verify the email address* after signing up
- Go to `/api/auth/login`, log in with the previously entered username and password
- You should see a page with an error message saying "unauthorized (Please verify your email before logging in.)"
- Go to `/api/auth/logout`, then to `/api/auth/login`, you should see the same "unauthorized" error message instead of the universal login page

Environment

- Version of this library used: 1.2.0
- Version of the platform or framework used, if applicable: node v12.18.3, nextJS v10.0.6
- Other relevant versions (language, server software, OS, browser): Auth0 Classic universal login with forced email verification

2

adamjmcgrath added the **needs investigation** label on 8 Apr 2021

adamjmcgrath commented on 9 Apr 2021 Member

Hi @Romainpaulus - thanks for raising this

This is expected behaviour - let me have a chat with my team about what changes, if any, we should make to this.

In the meantime, you can call the **logout endpoint** directly: `https://{AUTH0_DOMAIN}/v2/logout?returnTo={LOGOUT_URL}&client_id={CLIENT_ID}`

adamjmcgrath added **question** and removed **needs investigation** labels on 9 Apr 2021

Figure 15. First case: answer to the issue by a member of the repository

Detect When Client Closes Client-Side Streaming Call #1145

Closed quisc opened this issue on 6 Mar 2021 · 1 comment · Fixed by #1147

quisc commented on 6 Mar 2021 Contributor

What are you trying to achieve?

I need to detect server-side inside a client-side streaming call handler function when a client disconnects from the streaming call in order to clean up state / free resources and subscriptions.

What have you tried so far?

For server-side streaming calls I can subscribe to `context.statusPromise.futureResult` where `context` is a `StreamingResponseCallContext`. This seems to always be called eventually, even if the client (in our case an iOS app) loses connection / is killed. However, for client-side streaming calls the context is of type `UnaryResponseCallContext` which doesn't seem to have a corresponding future that I can subscribe to.

How do I get a callback when the client-side streaming call is closed / connection is lost?

quisc added the **question** label on 6 Mar 2021

glbrntt commented on 8 Mar 2021 Collaborator

This isn't possible at the moment. I think we'll have to add some API to the context here so that you can be notified when the RPC is torn down. It'd be best if we add this for all RPC types so there's a clear and consistent way to do this.

1

Assignees
 No one assigned

Labels
enhancement **question**

Projects
 None yet

Milestone
 No milestone

Development
 Successfully merging a pull request
 Add a 'closeFuture' to the `UnaryResponseCallContext` in `gibbrntt/grpc-swift`

Notifications
Subscribe
 You're not receiving notifications

2 participants

Figure 16. Second case: an issue labeled as question, which is then labeled as enhancement by a repository collaborator

This shows how different teams may use labels differently: the issue is objectively a question, but the collaborator decided to use that question as a reminder or a starting point to enhance the repository including the feature described by the question. And so, he labeled the issue as enhancement. But from the text, how are we supposed to understand that that feature is not present in the code and the collaborator will be interested in adding it? We cannot, and we believe that classifying this issue as a question is correct. From this example we also understand that we cannot trust random users in random repositories to train an issue classifier.

6.2 Handling noise

Inspecting the dataset, we found out that the data was really noisy. We saw in Section 6, specifically in Figure 16, how issue labels can be interpreted differently. As we can see from Table 6, in the dataset there are lots of projects with only one issue. This could mean that:

- the project is inactive, so there are few issues,
- the project is personal, so there is a single developer which is using the issue tracker as a reminder for future development of his project.

As suggested in [86], the majority of the projects are personal and inactive. So, mining GitHub issues without some filters to avoid those kind of projects, is not a good choice. If you have a project and you use it alone, there's no need to agree on the issue label with others, since the only one who takes care of the project is the owner. If issues are not used for a team to cooperate, probably there isn't a well established rationale on how to label issues. So we decided to filter out projects with only one issue in the dataset.

Table 6. Distribution of repositories with only one issue and with more issues in the dataset

issues per repo	no. repos
= 1	52.348
> 1	75.247
total	127.595

Many studies consider the number of stars of a project to be a proxy of their quality [87, 88]. As we show in Table 7, there are a lot of low quality repositories in the dataset. We believe that using higher quality projects as training set for a model, which are repositories making active use of the issue tracking system to work on the project, we can obtain better quality predictions.

Both of our filters are aimed to exclude small and low quality projects, which don't make use of issue tracker often and probably when they do, they do not use it in the right way. It's not important to be able to predict the label of an issue which comes from a small repository or from personal project: a small repository will not have a huge number of issues to handle. Prioritizing issues is something useful for bigger teams, since they receive high numbers of issues. So, we decided to keep only projects with at least two stars.

Table 7. Distribution of repositories with at most two stars or more than two stars in the dataset

stars	no. repos
≤ 1	46.470
> 1	81.125
total	127.595

To this aim, we filtered out issues from both the training and the testing set, and then retrained a model using only the issues from projects with at least two stars, which had at least two issues in the dataset. We also removed duplicates and issues coming from projects which had been deleted from GitHub. After applying this filtering we obtain the dataset distribution shown in Table 8. As a result, we filtered out more than 200,000

Table 8. Filtered dataset with label distribution.

	Train set		Test set	
bug	290,801	(55%)	33,275	(54%)
enhancement	188,643	(35%)	21,584	(35%)
question	53,361	(10%)	6,454	(11%)
total	532,805		61,313	

We train the issue classifier with the filtered dataset and obtain the performances shown in Table 9. Although the test set is different, there are not relevant changes in the micro F1, which is substantially the same. The F1 for the question and the bug class are higher while for the enhancement class is lower. Still the changes are very small and so we cannot assert that our filtering was useful in some way. The recall for the question class remains critical, which can be justified once again by the lower number of examples representing that class.

We then start conjecturing that maybe, in older repositories, it is more probable that a labeling rationale has been established, while for newer repositories we could have more noise, since the project is young and needs still time to draw up some guidelines for labeling. So, we set up another filter:

- remove projects with age less than one year

Table 9. Performance on the filtered dataset.

Class	Prec	Rec	F1
bug	.8771	.9095	.8930
enhanc.	.8562	.8474	.8517
question	.6808	.5744	.6231
micro avg	.8524	.8524	.8524
macro avg	.8074	.7771	.7893

- split the remaining projects in two ranges, $[1, 4]$ years and $]4, +\infty)$

This kind of split is inspired by [89]. This filter is applied after the ones described above. We show the distribution of the two splits in Table 10. As we can see, in the dataset there are a lot more young projects than the old ones.

Table 10. Dataset filtered and split by year with label distribution.

Class	1-4 year		4+ year split	
	Train set	Test set	Train set	Test set
bug	197,003 (53%)	22,548 (53%)	93,798 (58%)	10,727 (58%)
enhancement	145,658 (39%)	16,609 (39%)	42,985 (27%)	4,975 (27%)
question	29,957 (8%)	3,714 (9%)	23,404 (15%)	2,740 (15%)
total	372,618	42,871	160,187	18,442

We then retrain the RoBERTa model for both the splits and obtain the performance shown in Table 11.

Table 11. Performance on the filtered dataset, on the years split.

Class	1-4 year				4+ year split			
	Prec	Rec	F1	Supp	Prec	Rec	F1	Supp
bug	.8831	.9080	.8953	22,548	.8694	.8970	.8830	10,727
enhanc.	.8646	.8677	.8661	16,609	.8082	.7735	.7905	4,975
question	.6664	.5417	.5977	3,714	.6748	.6438	.6589	2,740
micro avg	.8606	.8606	.8606	42,871	.8260	.8260	.8260	18,442
macro avg	.8047	.7725	.7864	42,871	.7842	.7714	.7775	18,442

As we can see, the performance on the 1 – 4 year split are quite similar to the performances obtained on the original test set. For the 4+ year split, instead, we have a significant drop in the micro F1, while the classifier has a higher recall and F1 for the minority class. This drop can be explained by the low number of examples present in the second split. While the improvement for the prediction of the question class, is justified by the fact that in the 4+ year split we have a bigger proportion of the question class in the training set. We can

overall say that our filter weren't useful to improve the performances of our issue-classifier. We used filters which are simple, intuitive and commonly accepted. Anyways these procedures haven't led us to significative improvements in the issue classification task. This probably because of the consistent drop in the number of data remaining, especially after the two splits, and also the imbalance of the labels in the dataset.

7. Conclusions

We described our participation to the issue classification competition at the NLBSE'22 Workshop and some follow-up studies. We proposed a supervised approach for the automatic classification of GitHub issues as either bug, enhancement, or question. The two supervised approaches we proposed leverage transformer-based contextual embeddings and both outperform the baseline set by the competition organizers. As a result of our study and also of the tool competition, we can say that BERT and his variants are actually state-of-the-art for the issue report classification task. The performances in the Tool Competition were all quite similar, and the changes in the overall micro F1 were lower than 0.06. We did not conduct hyperparameter tuning to find the best parameters for the model when facing this task. This because fine-tuning BERT model is computational intensive, and requires a GPU. It may take a very long time to perform an extensive hyperparameter tuning without having adequate resources. We believe that tuning hyperparameters could increase the performances of our model. Although Ticket Tagger is the last model in the performances ranking, it can be still useful because of his speed and lightness. Actually it probably is the model which is easier to deploy in terms of potential spent resources and availability. We discovered a potential vulnerability of the dataset, which is big amount of noise. The dataset unbalance and the noise are probably the reasons why we couldn't achieve good performance in recognizing issues labeled as question. We tried with different attempt to filter out noise from the dataset, such as small and young repositories, which cannot be trusted blindly. Still we haven't obtained encouraging results in this direction.

7.1 Future Works

As future work, we plan to further investigate this task and this dataset:

- hyperparameter tuning, which is necessary to find the best set of hyperparameter to maximize the performances observed on the test set. Results in this sense may guide future research in this area and give more validity to our study.
- compare the performance of BERT-based models with traditional machine learning (such as SVM, Naive Bayes, etc.). This also enables a comparison based on the inference time, which we expect to be lower on traditional machine learning approaches.
- experiment with cross/within project settings, to understand if it is useful to fine-

tuned a model on the specific project or it is better to have a training set composed of more, reliable projects.

- try to approach the same task with a different dataset, such as the JIRA dataset presented in [33]. This dataset is composed of big projects which could be considered reliable, but still have an unbalanced distribution. Experimenting on other datasets is important to understand whether our findings still hold with different dataset, or having a different dataset can lead to a better/worse model. It is also useful for understanding which dataset one should use when building a model which can be deployed and used in real scenarios.

List of Figures

1	Bugzilla, list of bugs in the KDE project, from [19]	5
2	An issue tracked by JIRA, from [25]	7
3	An issue tracked by GitLab, from [34]	8
4	Illustration of Transformers architecture, from [56]	11
5	Illustration of the dependencies encoded by the self-attention layers . As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" – in a sense, the model’s representation of the word "it" bakes in some of the representation of both "animal" and "tired", from [57]	13
6	Different variants of BERT, from [59].	14
7	BERT creating word embeddings, from [59].	14
8	Example of GitHub issue, from [64].	16
9	Custom labels in GitHub Issues, from [67].	17
10	Tool Competition page of the NLBSE’22 workshop	18
11	The two classifiers implemented for issue labeling.	24
12	Illustration of the process adopted to select the best BERT model	25
13	Concatenation of the BERT embedding with the issue author association one-hot encoding vector	27
14	First case: an issue labeled as question, which actually contains a bug report	35
15	First case: answer to the issue by a member of the repository	36
16	Second case: an issue labeled as question, which is then labeled as enhancement by a repository collaborator	36

List of Tables

1	Dataset with label distribution.	21
2	Model selection: the best performance achieved on the validation set for all fine-tuned models.	26
3	Performance of the system on the test.	29
4	Issue classification results for bugs, enhancements and questions. The models are ranked by Avg., the micro average precision/recall/F1-score over the three issue types.	31
5	Confusion matrix on the test set for Classifier 1	34
6	Distribution of repositories with only one issue and with more issues in the dataset	37
7	Distribution of repositories with at most two stars or more than two stars in the dataset	38
8	Filtered dataset with label distribution.	38
9	Performance on the filtered dataset.	39
10	Dataset filtered and split by year with label distribution.	39
11	Performance on the filtered dataset, on the years split.	39

References

- [1] Giuliano Antoniol et al. “Is It a Bug or an Enhancement? A Text-Based Approach to Classify Change Requests”. In: *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. CASCON '08. Ontario, Canada: Association for Computing Machinery, 2008. ISBN: 9781450378826. DOI: 10.1145/1463788.1463819. URL: <https://doi.org/10.1145/1463788.1463819>.
- [2] Kim Herzig, Sascha Just, and Andreas Zeller. “It’s not a bug, it’s a feature: How misclassification impacts bug prediction”. In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 392–401. DOI: 10.1109/ICSE.2013.6606585.
- [3] Nitish Pandey et al. “Automated classification of software issue reports using machine learning techniques: an empirical study”. In: *Innovations in Systems and Software Engineering* 13 (Dec. 2017). DOI: 10.1007/s11334-017-0294-1.
- [4] Jun Wang, Xiaofang Zhang, and Lin Chen. “How well do pre-trained contextual language representations recommend labels for GitHub issues?” In: *Knowledge-Based Systems* 232 (2021), p. 107476. ISSN: 0950-7051. DOI: <https://doi.org/10.1016/j.knosys.2021.107476>. URL: <https://www.sciencedirect.com/science/article/pii/S0950705121007383>.
- [5] M. Izadi, K. Akbari, and A. Heydarnoori. “Predicting the objective and priority of issue reports in software repositories.” In: *Empir Software Eng* 27 (2022). ISSN: 0950-7051. DOI: 10.1007/s10664-021-10085-3. URL: <https://link.springer.com/article/10.1007/s10664-021-10085-3>.
- [6] Rafael Kallis et al. “Predicting issue types on GitHub”. In: *Science of Computer Programming* 205 (2021), p. 102598. ISSN: 0167-6423. DOI: <https://doi.org/10.1016/j.scico.2020.102598>. URL: <https://www.sciencedirect.com/science/article/pii/S0167642320302069>.
- [7] Rafael Kallis et al. “Ticket Tagger: Machine Learning Driven Issue Classification”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 406–409. DOI: 10.1109/ICSME.2019.00070.

- [8] Armand Joulin et al. “Bag of Tricks for Efficient Text Classification”. In: *Proc. of the 15th Conf. of the European Chapter of the Association for Computational Linguistics*. Valencia, Spain: ACL, Apr. 2017, pp. 427–431. URL: <https://aclanthology.org/E17-2068>.
- [9] Rafael Kallis et al. “NLBSE’22 Tool Competition”. In: *Proceedings of The 1st International Workshop on Natural Language-based Software Engineering (NLBSE’22)*. 2022.
- [10] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].
- [11] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL].
- [12] Zhenzhong Lan et al. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020. arXiv: 1909.11942 [cs.CL].
- [13] Colavito Giuseppe, Lanubile Filippo, and Novielli Nicole. *Issue-Report-Classification-Using-RoBERTa*. Version 1.0.0. Mar. 2022. URL: <https://github.com/collab-uniba/Issue-Report-Classification-Using-RoBERTa>.
- [14] D.E. Perry, N.A. Staudenmayer, and L.G. Votta. “People, organizations, and process improvement”. In: *IEEE Software* 11.4 (1994), pp. 36–45. DOI: 10.1109/52.300082.
- [15] Yunwen Ye. “Supporting software development as knowledge-intensive and collaborative activity”. In: *Proceedings of The IEEE - PIEEE* (Jan. 2006). DOI: 10.1145/1137661.1137666.
- [16] Thomas Zimmermann et al. “What Makes a Good Bug Report?” In: *IEEE Transactions on Software Engineering* 36.5 (2010), pp. 618–643. DOI: 10.1109/TSE.2010.63.
- [17] Dane Bertram et al. “Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams”. In: Jan. 2010, pp. 291–300. DOI: 10.1145/1718918.1718972.
- [18] Walid Maalej Clara Marie Lüders Abir Bouraffa. “Beyond Duplicates: Towards Understanding and Predicting Link Types in Issue Tracking Systems”. In: (July 2022).
- [19] Wikipedia. *Bugzilla*. URL: <https://it.wikipedia.org/wiki/Bugzilla>.
- [20] J. Janák. “Issue tracking systems”. In: *Brno, spring* (2009), p. 17.

- [21] Robert Sandusky and Les Gasser. “Negotiation and the coordination of information and activity in distributed software problem management”. In: Jan. 2005, pp. 187–196. DOI: 10.1145/1099203.1099238.
- [22] Christine Halverson et al. “Designing task visualizations to support the coordination of work in software development”. In: Jan. 2006, pp. 39–48. DOI: 10.1145/1180875.1180883.
- [23] John Anvik, Lyndon Hiew, and Gail Murphy. “Who should fix this bug?” In: vol. 2006. May 2006, pp. 361–370. DOI: 10.1145/1134336.
- [24] N.E. Fenton and M. Neil. “A critique of software defect prediction models”. In: *IEEE Transactions on Software Engineering* 25.5 (1999), pp. 675–689. DOI: 10.1109/32.815326.
- [25] Atlassian. *What is an issue in JIRA*. URL: <https://support.atlassian.com/jira-software-cloud/docs/what-is-an-issue/>.
- [26] Neil A. Ernst and Gail C. Murphy. “Case studies in just-in-time requirements analysis”. In: *2012 Second IEEE International Workshop on Empirical Requirements Engineering (EmpiRE)*. 2012, pp. 25–32. DOI: 10.1109/EmpIRE.2012.6347678.
- [27] Ahmed Lamkanfi et al. “Comparing Mining Algorithms for Predicting the Severity of a Reported Bug”. In: *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*. CSMR ’11. USA: IEEE Computer Society, 2011, pp. 249–258. ISBN: 9780769543437. DOI: 10.1109/CSMR.2011.31. URL: <https://doi.org/10.1109/CSMR.2011.31>.
- [28] Ahmed Lamkanfi et al. “Predicting the severity of a reported bug”. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010, pp. 1–10. DOI: 10.1109/MSR.2010.5463284.
- [29] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. “Improving Bug Triage with Bug Tossing Graphs”. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE ’09. Amsterdam, The Netherlands: Association for Computing Machinery, 2009, pp. 111–120. ISBN: 9781605580012. DOI: 10.1145/1595696.1595715. URL: <https://doi.org/10.1145/1595696.1595715>.

- [30] Jayati Deshmukh et al. “Towards Accurate Duplicate Bug Retrieval Using Deep Learning Techniques”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 115–124. DOI: 10.1109/ICSME.2017.69.
- [31] Jianjun He et al. “Duplicate Bug Report Detection Using Dual-Channel Convolutional Neural Networks”. In: *Proceedings of the 28th International Conference on Program Comprehension*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 117–127. ISBN: 9781450379588. URL: <https://doi.org/10.1145/3387904.3389263>.
- [32] Xiaoyin Wang et al. “An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information”. In: *Proceedings of the 30th International Conference on Software Engineering. ICSE '08*. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 461–470. ISBN: 9781605580791. DOI: 10.1145/1368088.1368151. URL: <https://doi.org/10.1145/1368088.1368151>.
- [33] Lloyd Montgomery, Clara Lüders, and Walid Maalej. “Jira: An Alternative Issue Tracking Dataset”. In: (Jan. 2022).
- [34] GitLab. *101 - No Tissues with Issues*. URL: <https://about.gitlab.com/handbook/marketing/strategic-marketing/getting-started/101/>.
- [35] Y. Yang and T. Joachims. “Text categorization”. In: *Scholarpedia* 3.5 (2008). revision #137225, p. 4242. DOI: 10.4249/scholarpedia.4242.
- [36] Fabrizio Sebastiani. “Machine Learning in Automated Text Categorization”. In: *ACM Comput. Surv.* 34.1 (Mar. 2002), pp. 1–47. ISSN: 0360-0300. DOI: 10.1145/505282.505283. URL: <https://doi.org/10.1145/505282.505283>.
- [37] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. “Understanding bag-of-words model: a statistical framework”. In: *International Journal of Machine Learning and Cybernetics* 1 (2010), pp. 43–52.
- [38] Santiago González-Carvajal and Eduardo C. Garrido-Merchán. *Comparing BERT against traditional machine learning text classification*. 2020. DOI: 10.48550/ARXIV.2005.13012. URL: <https://arxiv.org/abs/2005.13012>.
- [39] Karen Spärck Jones. “A statistical interpretation of term specificity and its application in retrieval”. In: *J. Documentation* 60 (2004), pp. 493–502.

- [40] Thomas K Landauer, Peter W. Foltz, and Darrell Laham. “An introduction to latent semantic analysis”. In: *Discourse Processes* 25.2-3 (1998), pp. 259–284. DOI: 10.1080/01638539809545028. eprint: <https://doi.org/10.1080/01638539809545028>. URL: <https://doi.org/10.1080/01638539809545028>.
- [41] Xiang Zhang, Junbo Zhao, and Yann LeCun. *Character-level Convolutional Networks for Text Classification*. 2015. DOI: 10.48550/ARXIV.1509.01626. URL: <https://arxiv.org/abs/1509.01626>.
- [42] Frederico Souza and João Filho. *BERT for Sentiment Analysis: Pre-trained and Fine-Tuned Alternatives*. 2022. DOI: 10.48550/ARXIV.2201.03382. URL: <https://arxiv.org/abs/2201.03382>.
- [43] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. DOI: 10.48550/ARXIV.1301.3781. URL: <https://arxiv.org/abs/1301.3781>.
- [44] Shervin Minaee et al. *Deep Learning Based Text Classification: A Comprehensive Review*. 2020. DOI: 10.48550/ARXIV.2004.03705. URL: <https://arxiv.org/abs/2004.03705>.
- [45] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. *A Convolutional Neural Network for Modelling Sentences*. 2014. DOI: 10.48550/ARXIV.1404.2188. URL: <https://arxiv.org/abs/1404.2188>.
- [46] Alexis Conneau et al. *Very Deep Convolutional Networks for Text Classification*. 2016. DOI: 10.48550/ARXIV.1606.01781. URL: <https://arxiv.org/abs/1606.01781>.
- [47] Rie Johnson and Tong Zhang. “Deep Pyramid Convolutional Neural Networks for Text Categorization”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 562–570. DOI: 10.18653/v1/P17-1052. URL: <https://aclanthology.org/P17-1052>.
- [48] Yizhe Zhang et al. *Deconvolutional Paragraph Representation Learning*. 2017. DOI: 10.48550/ARXIV.1708.04729. URL: <https://arxiv.org/abs/1708.04729>.
- [49] Dinghan Shen et al. “Deconvolutional Latent-Variable Model for Text Sequence Matching”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 2018). DOI: 10.1609/aaai.v32i1.11991. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/11991>.

- [50] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. *Recurrent Neural Network for Text Classification with Multi-Task Learning*. 2016. DOI: 10.48550/ARXIV.1605.05101. URL: <https://arxiv.org/abs/1605.05101>.
- [51] Dani Yogatama et al. *Generative and Discriminative Text Classification with Recurrent Neural Networks*. 2017. DOI: 10.48550/ARXIV.1703.01898. URL: <https://arxiv.org/abs/1703.01898>.
- [52] Minjoon Seo et al. *Neural Speed Reading via Skim-RNN*. 2017. DOI: 10.48550/ARXIV.1711.02085. URL: <https://arxiv.org/abs/1711.02085>.
- [53] Zichao Yang et al. “Hierarchical Attention Networks for Document Classification”. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, June 2016, pp. 1480–1489. DOI: 10.18653/v1/N16-1174. URL: <https://aclanthology.org/N16-1174>.
- [54] Zhouhan Lin et al. *A Structured Self-attentive Sentence Embedding*. 2017. DOI: 10.48550/ARXIV.1703.03130. URL: <https://arxiv.org/abs/1703.03130>.
- [55] Chi Sun et al. *How to Fine-Tune BERT for Text Classification?* 2019. DOI: 10.48550/ARXIV.1905.05583. URL: <https://arxiv.org/abs/1905.05583>.
- [56] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].
- [57] Jay Alammar. *The Illustrated Transformer*. URL: <https://jalammar.github.io/illustrated-transformer/>.
- [58] University of Harvard. *The Annotated Transformer*. URL: <http://nlp.seas.harvard.edu/2018/04/03/attention.html>.
- [59] Jay Alammar. *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning)*. URL: <https://jalammar.github.io/illustrated-bert/>.
- [60] Matthew E. Peters et al. *Deep contextualized word representations*. 2018. arXiv: 1802.05365 [cs.CL].
- [61] Tegawendé F. Bissyandé et al. “Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub”. In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 2013, pp. 188–197. DOI: 10.1109/ISSRE.2013.6698918.

- [62] Sebastiano Panichella et al. “How Developers’ Collaborations Identified from Different Sources Tell Us about Code Changes”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 251–260. DOI: 10.1109/ICSME.2014.47.
- [63] Qiang Fan et al. “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)*. 2017, pp. 121–130. DOI: 10.1109/ESEM.2017.19.
- [64] GeoNode. *Work With GitHub Issues and Pull Requests*. URL: https://doc-geonode.readthedocs.io/en/latest/005_dev_workshop/008_contribute_geonode/work_with_github.html.
- [65] Javier Luis Cánovas Izquierdo et al. “GiLA: GitHub label analyzer”. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 479–483. DOI: 10.1109/SANER.2015.7081860.
- [66] Zhifang Liao et al. “Exploring the characteristics of issue-related behaviors in GitHub using visualization techniques”. English. In: *IEEE Access* 6 (2018). Acceptance from VoR OA article however no CC licence on article (see p1 of VoR). Applied ’no exception’ as article doesn’t meet our definition for Gold exception. ET 14/1/20 ET, pp. 24003–24015. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2810295.
- [67] Steve Gordon. *WORKING ON YOUR FIRST GITHUB ISSUE*. URL: <https://www.stevejgordon.co.uk/working-on-your-first-github-issue>.
- [68] Yu Zhou et al. “Combining Text Mining and Data Mining for Bug Report Classification”. In: *2014 IEEE International Conference on Software Maintenance and Evolution* (2014), pp. 311–320.
- [69] Rafael Kallis et al. “NLBSE’22 Tool Competition”. In: *Proceedings of The 1st International Workshop on Natural Language-based Software Engineering (NLBSE’22)*. 2022.
- [70] Ilya Grigorik. *GH Archive*. URL: <https://www.gharchive.org/>.
- [71] *Google BigQuery*. URL: <https://cloud.google.com/bigquery/>.
- [72] Marco Polignano et al. “ALBERTO: Italian BERT Language Understanding Model for NLP Challenging Tasks Based on Tweets”. In: Nov. 2019.

- [73] Marco Polignano et al. “Lexicon Enriched Hybrid Hate Speech Detection with Human-Centered Explanations”. In: UMAP ’22 Adjunct. Barcelona, Spain: Association for Computing Machinery, 2022, pp. 184–191. ISBN: 9781450392327. DOI: 10.1145/3511047.3537688. URL: <https://doi.org/10.1145/3511047.3537688>.
- [74] Marco Pota et al. “An Effective BERT-Based Pipeline for Twitter Sentiment Analysis: A Case Study in Italian”. In: *Sensors* 21.1 (2021). ISSN: 1424-8220. DOI: 10.3390/s21010133. URL: <https://www.mdpi.com/1424-8220/21/1/133>.
- [75] Christos Baziotis, Nikos Pelekis, and Christos Doukeridis. “DataStories at SemEval-2017 Task 4: Deep LSTM with Attention for Message-level and Topic-based Sentiment Analysis”. In: *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. Vancouver, Canada: Association for Computational Linguistics, Aug. 2017, pp. 747–754. DOI: 10.18653/v1/S17-2126. URL: <https://aclanthology.org/S17-2126>.
- [76] Nguyen Manh Duc Tuan and Pham Quang Nhat Minh. *Multimodal Fusion with BERT and Attention Mechanism for Fake News Detection*. 2021. DOI: 10.48550/ARXIV.2104.11476. URL: <https://arxiv.org/abs/2104.11476>.
- [77] Maliheh Izadi. *CatIss: An Intelligent Tool for Categorizing Issues Reports using Transformers*. 2022. DOI: <https://doi.org/10.1145/3528588.3528662>. URL: <https://arxiv.org/pdf/2203.17196.pdf>.
- [78] Shikhar Bharadwaj and Tushar Kadam. *GitHub Issue Classification Using BERT-Style Models*. 2022.
- [79] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. DOI: 10.48550/ARXIV.2002.08155. URL: <https://arxiv.org/abs/2002.08155>.
- [80] Zhilin Yang et al. *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. 2019. DOI: 10.48550/ARXIV.1906.08237. URL: <https://arxiv.org/abs/1906.08237>.
- [81] Mohammed Latif Siddiq and Joanna C.S. Santos. *BERT-Based GitHub Issue Report Classification*. 2022.
- [82] Alexander Trautsch and Steffen Herbold. *Predicting Issue Types with seBERT*. 2022.

- [83] Julian von der Mosel, Alexander Trautsch, and Steffen Herbold. *On the validity of pre-trained transformers for natural language processing in the software engineering domain*. 2021. DOI: 10.48550/ARXIV.2109.04738. URL: <https://arxiv.org/abs/2109.04738>.
- [84] Giuseppe Colavito, Filippo Lanubile, and Nicole Novielli. *Issue Report Classification Using Pre-trained Language Models*. 2022. DOI: <https://doi.org/10.1145/3528588.3528659>.
- [85] Sebastian Hofstätter and Allan Hanbury. *Let's measure run time! Extending the IR replicability infrastructure to include performance aspects*. 2019. DOI: 10.48550/ARXIV.1907.04614. URL: <https://arxiv.org/abs/1907.04614>.
- [86] Eirini Kalliamvakou et al. “The Promises and Perils of Mining GitHub”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 92–101. ISBN: 9781450328630. DOI: 10.1145/2597073.2597074. URL: <https://doi.org/10.1145/2597073.2597074>.
- [87] Sumon Biswas et al. “Boa Meets Python: A Boa Dataset of Data Science Software in Python Language”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 577–581. DOI: 10.1109/MSR.2019.00086.
- [88] Nuthan Munaiah et al. “Curating GitHub for engineered software projects”. In: (Dec. 2016). DOI: 10.7287/PEERJ.PREPRINTS.2617.
- [89] Bogdan Vasilescu et al. “Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 401–405. DOI: 10.1109/ICSME.2014.62.