# Ticket Tagger: Machine Learning driven Ticket Classification

Rafael Kallis
rk@rafaelkallis.com

Philip Hofmann

Selin Fabel

Dominik Bünzli

Manpreet Singh

Nicolas Gordillo

## Abstract

Software maintenance is crucial: code should be kept up-to-date and error-free with little effort. In this context, issue trackers are an essential tool for creating, managing and reporting tickets or potential problems in software repositories. In this work, we select a model suitable for assigning labels to tickets on issue trackers hosted on GitHub, i.e., determines if a ticket is a bug report, a feature request or a question.
In order to improve the model's performance, We conduct an extensive experimental evaluation. Finally, we release the Ticket Tagger application which can be used publicly. The presented tool can help maintainers to keep control of their work-load and in the end, speeds up the whole maintenance process.

## 1 Introduction

In the life cycle of software, maintenance is a crucial topic. It is an issue with many aspects. First of all, the source code should be kept up-to-date and any potential flaws in terms of performance and correctness removed. On the other hand, a maintainer must invest as little time and effort as possible to solve the mentioned tasks to keep the software maintenance costs low. [5]

Important tools for maintainers are represented by issue tracking systems. Here, they report tickets or potential problems, manage them and keep track of their progress. But as useful they might be, many developers still end up with a rapidly growing workload and loose control of it. By means of machine learning techniques, we thus like to help maintainers to prioritize their workload, regain control of it and in the end, speed up the whole maintenance process.

The contributions of this paper are two-fold. First we perform a benchmark seeking a suitable machine learning model for labeling issue tracker tickets, i.e., given a recently opened issue, the model should predict if the ticket is a bug report, a feature request or a question. Having selected an appropriate model, we then perform an extensive experimental evaluation towards optimizing the train set in order to improve the predictive ability of our model.

The second contribution of our work is the release of an app that allows our trained model to be accessible to other software engineers. We develop a GitHub app that can be integrated with any new or existing software repository hosted on GitHub. When creating a new ticket on the repository's issue tracker, our app automatically assigns a relevant label based on our trained model.

The two models used for the experimentation are briefly described in 2. An overview of the experiments as well as the experimental evaluation is presented in Section 3.

## 2 Ticket Classification

Different methods can be used to extract labels for sentences. In our experimentation, we compare the accuracy of two models, the *Waikato Environment for Knowledge Analysis*, proposed by Garner [2], and *fastText* by Joulin et al. [4].

### 2.1 Weka

Developed at the University of Waikato, the Waikato Environment for Knowledge Analysis (Weka) is a data mining software implemented in Java [2]. It is used by machine learning researchers, industrial scientists or in the academic field to derive useful knowledge from large flat database files. For this purpose, Weka provides a collection of machine learning algorithms for data mining tasks such as preparation, classification, regression, clustering, association rules mining, and visualization. A large number of tools are implemented in Weka to support different approaches for the data mining tasks, like RandomForest and J48 Trees for classification.

### 2.2 fastText

Open sourced by Facebook AI Research in 2016, *fastText*, as proposed by Joulin et al., is a state-of-the-art text classifier providing both effective and efficient sentence classification [4]. The fastText classifier represents sentences as bag of words and uses several techniques to improve its accuracy and runtime performance, e.g., N-gram features to embed local word order information, a hierarchical softmax [3] to improve runtime during training and classification, etc. Joulin et al. also showed that whilst fastText's accuracy is competitive against several deep learning based models, it is several orders of magnitude faster for training and evaluation [4].

## 3 Experimental Evaluation

The goal of each classifier is to predict an issue label given text from a ticket. The models are trained on three labels: "bug", "enhancement" and "question". We chose these labels since they are included by default in every issue tracker on GitHub repositories and they encode the vast majority of tickets [1].

The experimental evaluation is structured as follows. We first perform a benchmark over the models introduced earlier, i.e., Weka and fastText. The best performing model will then be used for all subsequent experiments which involve optimizing the train set in order to improve the accuracy of the model. The idea is to select domain specific datasets in order to improve performance over that domain. Each proposed optimization is formulated and treated as a research question.

In general, we have a pair of datasets for each research question: an optimized and a baseline dataset. Both datasets in a pair contain an equal amount of entries and there is an even split across the labels.

As for each experiment, we run a 10-fold cross validation over the baseline and optimized dataset. During the process, we record *precision*, *recall* and the $F_1$ score.

The measures above are computed as follows:

$$\mathit{Precision} = \frac{\sum \text{true positives}}{\sum \text{true positives} + \sum \text{false positives}}$$

$$\mathit{Recall} = \frac{\sum \text{true positives}}{\sum \text{true positives} + \sum \text{false negatives}}$$

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

We now explain in detail the methodology used on the individual research questions and present the results of the evaluation process.

### 3.1 Model Selection

Our first experiment involves testing the performance of two models over a shared dataset. The models participating in the benchmark are Weka [2] and fastText [4]. We perform a 10-fold cross validation over the shared dataset containing pre-labeled tickets. Specifically, the dataset contains 10'000 bug reports, 10'000 feature requests and 10'000 questions. All tickets where drawn from GitHub at random[1] from GitHub,[2] a software repository hosting service with an integrated issue tracker.

Table 1 shows results of the 10-fold cross validation. We observe fastText having double digit advantages in every measure over Weka, which suggests that fastText is superior to Weka for the given dataset. Throughout the rest of the paper, we make exclusive use of fastText for all benchmarks.

In the following sections, we try optimizing the dataset in order to achieve a performance increase of fastText. We state a series of hypotheses and then consecutively test them in an extensive experimental evaluation.

[1]A ticket is drawn if it is closed during January 2018 and contains a label matching to one of the strings: "bug", "enhancement" or "question".

[2]https://github.com

| | | Weka | fastText | Δ |
|---|---|---|---|---|
| *Bug* | *Prec* | 58.3% | 82.2% | +23.9% |
| | *Rec* | 58.7% | 84.1% | +25.4% |
| | $F_1$ | 58.5% | 83.1% | **+24.6%** |
| *Enhancement* | *Prec* | 60.4% | 89.4% | +29.0% |
| | *Rec* | 63.2% | 76.3% | +13.1% |
| | $F_1$ | 61.7% | 82.3% | **+20.6%** |
| *Question* | *Prec* | 64.5% | 78.1% | +13.6% |
| | *Rec* | 61.1% | 87.4% | +26.3% |
| | $F_1$ | 62.8% | 82.5% | **+19.7%** |

Table 1: Model Benchmark: precision, recall and $F_1$ score of "bug", "enhancement" and "question" labels for the Weka and fastText models after a 10-fold cross validation.

## 3.2 Role of Spoken Language

In this section, we analyze if a spoken language-specific dataset affects the performance of our model. Our approach is to generate one dataset containing only English tickets, one baseline dataset with the same number of entries but drawn at random, and then compare the model's performance w.r.t. the two datasets.

| | | ENGLISH$_0$ | ENGLISH | Δ |
|---|---|---|---|---|
| *Bug* | *Prec* | 80.4% | 77.9% | -2.5% |
| | *Rec* | 67.4% | 76.8% | +9.4% |
| | $F_1$ | 73.3% | 77.3% | **+4.0%** |
| *Enhancement* | *Prec* | 72.7% | 79.0% | +6.3% |
| | *Rec* | 79.6% | 77.8% | -1.8% |
| | $F_1$ | 76.0% | 78.4% | **+2.4%** |
| *Question* | *Prec* | 74.8% | 76.8% | +2.0% |
| | *Rec* | 77.3% | 79.0% | +1.7% |
| | $F_1$ | 76.0% | 77.9% | **+1.9%** |

Table 2: Spoken Language-specific Dataset: precision, recall and $F_1$ score of "bug", "enhancement" and "question" labels for the baseline (ENGLISH$_0$) and optimized (ENGLISH) dataset after a 10-fold cross validation.

To generate the English dataset ENGLISH, we run a language classifier against tickets drawn at random from a pool of tickets. The language classifier used is a javascript port of "guess language",[3] which uses heuristics based on character sets and trigrams. The resulting optimized dataset contains 24'600, evenly split tickets. The baseline dataset ENGLISH$_0$ is created by drawing 24'600 tickets at random.

We benchmark our two datasets and present the results in Table 2. We observe that almost all measures increase when comparing ENGLISH against ENGLISH$_0$. This supports the hypothesis that a train and test set containing only tickets from a single spoken language, affects the model's performance for that language.

There is no sufficient evidence to claim that a spoken language-specific dataset *always* improves the model's performance, as we tested the hypothesis only on a single language.

## 3.3 Role of Software Repository

In this section, we investigate if our model performs differently, given the train and test set come exclusively from a single software repository.

To do so, we plan to find a software repository which is large enough, i.e., has enough labelled tickets and record the model performance in comparison to a random dataset. Visual studio code,[4] a modern text editor from Microsoft, offers at least 2'000 tickets for each of the three labels, i.e., "bug", "enhancement" and "question", on its repository.[5]

We fetch the ticket metadata, i.e., title, labels and body, using the GitHub rest api.[6] The resulting dataset, VSCODE, contains 6'000 tickets evenly split across the labels. The baseline dataset, VSCODE$_0$, is created by drawing 6'000 tickets at random from a pool of tickets with an even label split.

We benchmark our model against the baseline and optimized dataset and present the results in Table 3. We observe an increase in the model's performance in almost all measures. This supports our hypothesis that a software-repository specific dataset affects the model's performance. We believe specialized vocabulary is developed at individual repositories, e.g., the occurrence of application specific error codes in a ticket, might suggest that the ticket is more likely a bug report.

We cannot present sufficient evidence that a software repository-specific dataset *always* improves

[3]https://github.com/wooorm/franc

[4]https://github.com/microsoft/vscode

[5]At the time of writing, the labels were named `Bug`, `feature-request` and `*question`, respectively.

[6]https://developer.github.com/v3/issues/list-issues-for-a-repository

| | | VSCODE$_0$ | VSCODE | Δ |
|---|---|---|---|---|
| *Bug* | *Prec* | 67.1% | 84.0% | +16.9% |
| | *Rec* | 68.0% | 70.6% | +2.6% |
| | $F_1$ | 67.4% | 76.6% | **+9.2%** |
| *Enhancement* | *Prec* | 74.3% | 71.1% | -3.2% |
| | *Rec* | 68.7% | 75.1% | +6.4% |
| | $F_1$ | 71.4% | 73.0% | **+1.6%** |
| *Question* | *Prec* | 70.3% | 70.7% | +0.4% |
| | *Rec* | 71.9% | 77.9% | +6.0% |
| | $F_1$ | 71.0% | 74.1% | **+3.1%** |

Table 3: Repository-specific Dataset: precision, recall and $F_1$ score of "bug", "enhancement" and "question" labels for the baseline (VSCODE$_0$) and optimized (VSCODE) dataset after a 10-fold cross validation.

the model's performance. We do not have any observations that suggest so, nevertheless we can't eliminate the possibility of

## 3.4 Role of Code Snippets

Code snippets on tickets can be seen commonly. They usually contain source code, stack-traces or other software related artifacts. In this section we investigate if such snippets affect the model's predictive performance. To be more precise, we define a code snippet to be any piece of text enclosed in triple backticks (` ``` `), which is special syntax used by the markup language *Markdown*[7] to indicate snippets.

We observed that 10% of all tickets contain code snippets. More surprisingly, 90% of all tickets containing a code snippet are bug reports. This might imply that losing the snippets might affect the model's performance.

To test the hypothesis, we benchmark our model against two datasets, one with tickets drawn at random (NOSNIP$_0$), and one dataset containing tickets with no code snippets (NOSNIP). Both NOSNIP$_0$ and NOSNIP contain each 22'500 rows and the labels are evenly split.

The results of the 10-fold cross validation are presented in Table 4. We observe marginal differences between the two datasets. This might suggest that code snippets do no affect the model's performance significantly, or that code snippets can be removed without having any significant regressions on the model's performance.

We believe this is the case because the morphological structure of a human written sentence differs from that of a code snippet. Because of that, various word embeddings used by fastText are rendered useless for code snippets, and therefore their presence or absence does not affect the accuracy of the model.

This thesis requires further investigation and taking advantage of code snippets would unlock a potential for further improvements in label prediction of issue tracker tickets.

| | | NOSNIP$_0$ | NOSNIP | Δ |
|---|---|---|---|---|
| *Bug* | *Prec* | 74.9% | 75.5% | +0.6% |
| | *Rec* | 76.9% | 73.8% | -3.1% |
| | $F_1$ | 75.9% | 75.4% | **-0.3%** |
| *Enhancement* | *Prec* | 75.9% | 73.9% | -2.0% |
| | *Rec* | 78.1% | 80.6% | +2.5% |
| | $F_1$ | 77.0% | 77.1% | **+0.1%** |
| *Question* | *Prec* | 78.7% | 78.3% | -0.4% |
| | *Rec* | 71.5% | 72.0% | +0.5% |
| | $F_1$ | 75.0% | 75.0% | **+0.0%** |

Table 4: Code Snippet-free Dataset: precision, recall and $F_1$ score of "bug", "enhancement" and "question" labels for the baseline (NOSNIP$_0$) and code snippet-free dataset (NOSNIP) after a 10-fold cross validation.

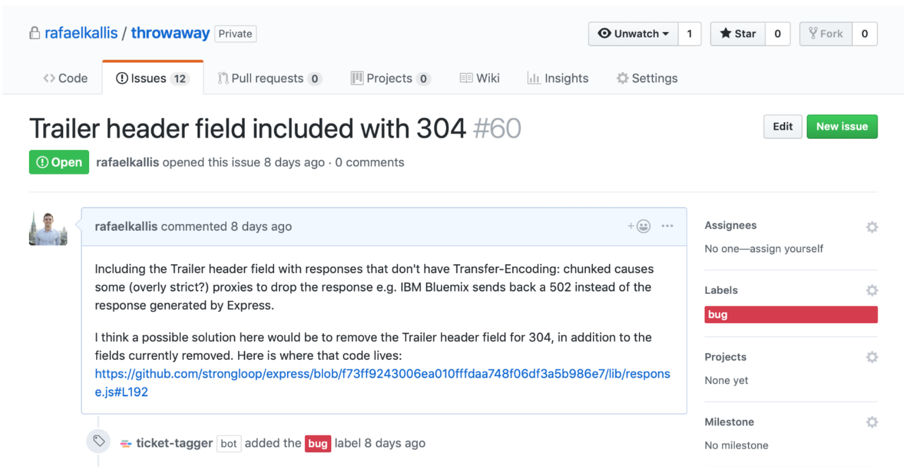
# 4 Application

One of our goals was to ease the maintenance process of other developers. We therefore integrated fastText into an app which we released to GitHub concurrent to the experimental evaluation.[8] The app connects to GitHub and automatically assigns labels to newly created tickets on software repositories. This not just rounds off our project but also gives fellow developers the opportunity to participate in our insights in a profitable way.

Our Ticket tagger application is freely accessible to any developer and can be integrated painlessly into existing GitHub repositories. Figure 1 depicts the standard tagging process. Each time, when a developer opens an issue on a software repository the application is triggered and runs a classification.

[7]https://guides.github.com/features/mastering-markdown

[8]https://github.com/apps/ticket-tagger



Trailer header field included with 304 #60

rafaelkallis opened this issue 8 days ago · 0 comments

rafaelkallis commented 8 days ago

Including the Trailer header field with responses that don't have Transfer-Encoding: chunked causes some (overly strict?) proxies to drop the response e.g. IBM Bluemix sends back a 502 instead of the response generated by Express.

I think a possible solution here would be to remove the Trailer header field for 304, in addition to the fields currently removed. Here is where that code lives: https://github.com/strongloop/express/blob/f73ff9243006ea010fffdaa748f06df3a5b986e7/lib/response.js#L192

ticket-tagger bot added the bug label 8 days ago

Assignees: No one—assign yourself

Labels: bug

Projects: None yet

Milestone: No milestone

Figure 1: Screenshot of Ticket Tagger functioning. Our app assigns the "bug" label to a newly created ticket.

In the example, the tagger correctly identifies the ticket as a bug report and automatically assigns the "bug" label to the ticket. The code owners can now immediately react to any urgent tickets, postpone less important issues such as feature requests or enhancements or transmit questions to the responsible persons.

## 5 Conclusion

In this work we selected a model suitable for assigning labels on issue trackers hosted on GitHub. The classification is performed by analyzing the text of the issues and determining whether the examined ticket is a bug report, a feature request or a question.

We then conducted an extensive experimental evaluation of our model in order to further improve its performance. The results show that our classifier allows to automatically assign labels with appreciable levels of precision and recall for all three categories. When investigating domain-specific datasets – which for example only contain data from a single spoken language or from a specific repository – we observed effects on the model's performance. Unfortunately, we were not able to find any statistical evidence. We do not know if domain-specific train sets yield robust performance improvements. For future work, we therefore plan a further investigation of this issue.

Concurrent to the experimental evaluation, we released our ticket tagger as an app on the GitHub market place. Fellow developers can integrate it into their repositories and with the automatic classification of their tickets speed up their maintenance process.

## References

[1] "Won't We Fix this Issue?" Qualitative Characterization and Automated Identification of Wontfix Issues on GitHub. To be published, 2018.

[2] S. R. Garner et al. Weka: The waikato environment for knowledge analysis. In *Proceedings of the New Zealand computer science research students conference*, pages 57–64, 1995.

[3] J. Goodman. Classes for fast maximum entropy training. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, volume 1, pages 561–564. IEEE, 2001.

[4] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[5] F. P. and V. H. H. How to save on software maintenance costs, omnext white paper. *SOURCE 2 VALUE*, 2010.