STACC: Code Comment Classification using SentenceTransformers

Al-Kaswan, A.; Izadi, M.; van Deursen, A.

DOI
10.1109/NLBSE59153.2023.00014

Publication date
2023

Document Version
Final published version

Published in
2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)

Citation (APA)

Important note
To cite this publication, please use the final published version (if applicable). Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights. We will remove access to the work immediately and investigate your claim.
Green Open Access added to TU Delft Institutional Repository

'You share, we take care!' - Taverne project

https://www.openaccess.nl/en/you-share-we-take-care

Otherwise as indicated in the copyright section: the publisher is the copyright holder of this work and the author uses the Dutch legislation to make this work public.
STACC: Code Comment Classification using SentenceTransformers

Ali Al-Kaswan  
*Delft University of Technology*  
Delft, The Netherlands  
am.al-kaswan@tudelft.nl

Maliheh Izadi  
*Delft University of Technology*  
Delft, The Netherlands  
m.izadi@tudelft.nl

Arie van Deursen  
*Delft University of Technology*  
Delft, The Netherlands  
am.vandeursen@tudelft.nl

I. INTRODUCTION

Code comments are an essential source of information for software developers. Code comments document functionalities, describe the usage, or are a place for developers to leave development notes, which help with maintainability [1] and ongoing development tasks [2]. Code comments also play a large role in helping developers understand source code [1].

Depending on the task at hand, not all types of code comments are useful. Especially when considering automated code tasks, like automatic code summarization [3], or code completion [4], only certain types of documentation are useful. Therefore, there is a need to classify to which category a piece of software documentation belongs.

In this tool paper, we present STACC: a set of SentenceTransformer-Assisted [5] Comment Classifiers. We make use of SetFit [6], an efficient and prompt-free framework for few-shot fine-tuning of Sentence Transformers. SetFit achieves high accuracy with little labelled data. The resulting models are also compact and fast to run.

STACC outperforms the baseline in every category in both recalls, as well as F1 scores. We achieve an average F1 score of 0.74 against the baseline of 0.31.

We make all of our models available on the HuggingFace Hub. The training Notebooks, which explain the model selection and training process in detail and show how to load and use STACC are available on GitHub and Google Colab. Finally, we also offer an interactive HuggingFace Space to experiment with our classifiers in the browser.

---

1HF Hub: https://huggingface.co/AISE-TUDelft
2STACC Repo: https://github.com/AISE-TUDelft/STACC
3STACC HF Space: https://huggingface.co/spaces/AISE-TUDelft/STACC

II. TOOL CONSTRUCTION

In this section, we provide details on how we constructed STACC. Specifically, how we selected the base model, and how we tuned the hyperparameters to create the set of classifiers.

A. Model Selection

We first test several candidate base models for our approach. We select the models from the Sentence-Transformers Sentence Embedding benchmark. We tune the models on a sample of 32 positive and negative examples from a single category, for 5 epochs using the standard settings in the SetFitTrainer. We select the hardest category, the one where the baseline performed poorest, to tune our hyperparameters. We measure the accuracy and F1 on the complete test set as well as the total runtime of the training and testing regime.

We first select two smaller models, paraphrase-MiniLM-L3-v2 which is specifically designed for semantic tasks and has a model size of only 61MB, all-MiniLM-L6-v2, a good all-around model, with a model size of 120MB. We also select alpnet-base-v2, which is the highest-performing model in the Sentence Embedding benchmark but is much larger with a model size of 420MB. Finally, we also selected st-codesearch-distilroberta-base, in the hope that its mixed training objectives, which included some code and documentation-related tasks would boost its performance. Its model size was 320MB. We attempt to run all-distilroberta-v1 but we find that this model was too big to run on our hardware with a model size of 1.4GB.

B. Hyperparameter Tuning

We tune the hyperparameters of the selected model trained on one category. We make use of the Optuna backend with SetFit to find the best hyperparameters. We tune the learning rate, the number of training epochs, the number of sentence-pairs generation iterations and the solver. We use make use of the entire dataset and run Optuna for 20 iterations to find the best hyperparameters. Finally, we use the selected base model and optimised hyperparameters to train a classifier for each of the 19 categories.

---

4Sentence-Transformers Model Overview: https://www.sbert.net/docs/pretrained_models.html#model-overview
models used for this task since data-hungry models cannot be trained with this amount of data.

B. Metrics

To score the classifiers we use the metrics specified by the competition as a function of a category \( c \in C \). Namely, the recall \( R_c \), precision \( P_c \), and F1 score \( F_{1,c} \). These scores are calculated per category \( c \). The F1 score is calculated as the geometric mean of the recall and precision, we additionally use the weighted F1 score, \( wF_{1,c} \) as well as the accuracy to also take the unbalanced nature of the data into account, but these scores are just used as a reference and are not taken into account for the final scoring. The metrics are defined as follows:

\[
\begin{align*}
\text{Accuracy}_c &= \frac{TP_c + TN_c}{TP_c + TN_c + FP_c + FN_c} \\
\text{Precision}_c &= \frac{TP_c}{TP_c + FP_c} \\
\text{Recall}_c &= \frac{TP_c}{TP_c + FN_c} \\
F_{1,c} &= 2 \cdot \frac{\text{Precision}_c \cdot \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}
\end{align*}
\]  

Finally, the competition submission score for \( M \) a set of models \( m_c \) against the baseline is defined as:

\[
\text{submission\_score}(M) = \frac{\sum_{c \in C} F_{1,c} \times 0.75}{|C|} + (\% \text{outperformed\_categories}) \times 0.25
\]  

This score takes the average F1-score achieved by the classifiers and weights it against the fraction of classifiers that outperform the baselines.

C. Baselines

The authors of the competition also provide baseline Random Forest Classifiers [8]. We observe quite poor recalls but decent wF1 scores across the board in Table IV. This indicates that the model tends to favour predicting non-membership for each of the classes, which the unbalanced nature of the data could explain. The category with the lowest F1 score is ‘Java Deprecation’, this is the category which we target for the model selection and Hyperparameter search.

D. Implementation

We use the HuggingFace SentenceTransformers and PyTorch packages to train STACC. The base model was loaded using the HuggingFace Hub. To process the data we use Pandas and the HuggingFace Dataset packages. The experiments are conducted on an Nvidia GTX 3080 GPU with 10GB of VRAM and an AMD Ryzen Threadripper 3990X paired with 128GB RAM.

IV. RESULTS

In this section, we discuss the results of our model search and hyperparameter tuning. We also compare our models to the baseline and provide a short discussion of the results.

---

3Code Comment Classification GitHub repo: https://github.com/nlbse2023/code-comment-classification
### Model Selection

The results of the model selection experiments are shown in Table II, and the training time is denoted in minutes. We can see that all-mpnet-base-v2 outperforms the other models by a decent margin. Its training time is longer but still workable. Unfortunately, the st-code search-distilroberta-base model did not seem to benefit from its pre-training regime despite being slower and larger than both MiniLM models.

### Hyperparameter Tuning

The importance of each hyperparameter is plotted in Figure 1. The importance of each hyperparameter is calculated using the fANOVA importance metric included in the Optuna backend of SetFit [9]. The obtained hyperparameters, the space which we searched and the base settings are shown in Table III. We find that the most important hyperparameter is the learning rate, followed by the number of training epochs. The importance of each hyperparameter is calculated using the fANOVA importance metric included in the Optuna backend of SetFit [9]. The obtained hyperparameters, the space which we searched and the base settings are shown in Table III. We find that the most important hyperparameter is the learning rate, followed by the number of training epochs.

### STACC Models

Finally, we use the selected base model and optimised hyperparameters to train a classifier for each of the 19 categories.

#### Table II: Accuracy, unbalanced F1 score and training time in minutes of different base models on the Java Depreciation category

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Accuracy</th>
<th>F1</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>paraphrase-MiniLM-L3-v2</td>
<td>0.88</td>
<td>0.435</td>
<td>1:22</td>
</tr>
<tr>
<td>all-MiniLM-L6-v2</td>
<td>0.89</td>
<td>0.453</td>
<td>2:30</td>
</tr>
<tr>
<td>all-mpnet-base-v2</td>
<td>0.93</td>
<td>0.646</td>
<td>4:37</td>
</tr>
<tr>
<td>st-code search-distilroberta-base</td>
<td>0.88</td>
<td>0.457</td>
<td>2:41</td>
</tr>
</tbody>
</table>

#### Table III: Hyperparameter search space and results

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Search space</th>
<th>Base</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>learning_rate</td>
<td>(1e-6, 1e-4)</td>
<td>2.00e-5</td>
<td>1.71e-05</td>
</tr>
<tr>
<td>Epochs</td>
<td>(1, 10)</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>LogReg fit iterations</td>
<td>(50, 300)</td>
<td>100</td>
<td>241</td>
</tr>
<tr>
<td>LogReg Solver</td>
<td>[newton-cg, lbfgs, lbblin]</td>
<td>lbblin</td>
<td>lbfgs</td>
</tr>
</tbody>
</table>

#### Figure 1: fANOVA importance of each hyperparameter

With these results, per Equation 2 we achieve a final submission score of:

\[
\text{submission score} = 0.74 \times 0.75 + 1 \times 0.25 = 0.81 \tag{3}
\]

#### D. Discussion

When comparing languages it becomes apparent that STACC tends to perform better on Java than both Python and Pharo. Across the languages, the average F1 scores are 0.84, 0.72 and 0.64.

For some categories, we see that our approach greatly outperforms the baseline. For instance, Java Deprecation and Python Summary, where the average F1 score is 0.86 and 0.54 higher, respectively.

While for others we observe that both our and the baseline classifiers perform quite poorly. Namely, Python Dependencies, Pharo Class references and Pharo Collaborators, which are very unbalanced and have relatively few positive instances. Another imbalanced category with few positive instances is Java Ownership, but our classifier performed admirably in this case, achieving a perfect F1 score. Looking at the data, we notice that each of the Java Ownership comments contains the token @author, which makes predicting these quite straightforward.

We also observed that the addition of the class name to the input of the model improves the average precision, recall, and F1 scores by 0.12, 0.02, and 0.09 respectively. The intuition is that the class name is in itself an extremely brief summary of the entire class. For brevity, we have omitted the results without the class names, but the fully trained Classifiers are also available on the HuggingFace Hub. Each classifier has two branches: ‘main’ for the regular model which expects inputs including the class name, and a branch tagged ‘V1’ for inputs without the class name.

While we focus on creating the best-performing classifiers for this task, a significant speedup can be achieved by reducing the number of training epochs. Instead of training for 6 epochs, tuning the models for 1 or 2 epochs will perform slightly worse than STACC’s classifiers while only requiring a fraction of the training time.

Besides just reducing the number of training epochs, the models could also be fine-tuned in a few-shot setting by taking a few positive and negative examples. In our evaluations, we

---

6STACC V1 example: https://huggingface.co/AISE-TUDelft/java-rational-classifier/tree/V1
TABLE IV: Performance of STACC against the baseline

<table>
<thead>
<tr>
<th>Language</th>
<th>Category</th>
<th>Baseline</th>
<th>STACC</th>
<th>ΔF₁,c,c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Pₐ,c</td>
<td>Rₐ,c</td>
<td>F₁,c,c</td>
</tr>
<tr>
<td>Java</td>
<td>Deprecation</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Java</td>
<td>Expand</td>
<td>0.35</td>
<td>0.27</td>
<td>0.30</td>
</tr>
<tr>
<td>Java</td>
<td>Ownership</td>
<td>1.00</td>
<td>0.68</td>
<td>0.81</td>
</tr>
<tr>
<td>Java</td>
<td>Pointer</td>
<td>0.67</td>
<td>0.24</td>
<td>0.35</td>
</tr>
<tr>
<td>Java</td>
<td>Rational</td>
<td>0.63</td>
<td>0.30</td>
<td>0.40</td>
</tr>
<tr>
<td>Java</td>
<td>Summary</td>
<td>0.38</td>
<td>0.29</td>
<td>0.33</td>
</tr>
<tr>
<td>Java</td>
<td>Usage</td>
<td>0.54</td>
<td>0.36</td>
<td>0.43</td>
</tr>
<tr>
<td>Pharo</td>
<td>Classref</td>
<td>0.33</td>
<td>0.06</td>
<td>0.10</td>
</tr>
<tr>
<td>Pharo</td>
<td>Collab</td>
<td>0.47</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>Pharo</td>
<td>Example</td>
<td>0.77</td>
<td>0.43</td>
<td>0.55</td>
</tr>
<tr>
<td>Pharo</td>
<td>Intent</td>
<td>0.58</td>
<td>0.33</td>
<td>0.42</td>
</tr>
<tr>
<td>Pharo</td>
<td>Keyimpl</td>
<td>0.18</td>
<td>0.10</td>
<td>0.13</td>
</tr>
<tr>
<td>Pharo</td>
<td>Keymsg</td>
<td>0.31</td>
<td>0.16</td>
<td>0.21</td>
</tr>
<tr>
<td>Pharo</td>
<td>Resp</td>
<td>0.59</td>
<td>0.33</td>
<td>0.43</td>
</tr>
<tr>
<td>Python</td>
<td>Devnotes</td>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>Python</td>
<td>Expand</td>
<td>0.26</td>
<td>0.20</td>
<td>0.22</td>
</tr>
<tr>
<td>Python</td>
<td>Parameters</td>
<td>0.51</td>
<td>0.22</td>
<td>0.31</td>
</tr>
<tr>
<td>Python</td>
<td>Summary</td>
<td>0.12</td>
<td>0.08</td>
<td>0.09</td>
</tr>
<tr>
<td>Python</td>
<td>Usage</td>
<td>0.47</td>
<td>0.18</td>
<td>0.26</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>0.44</td>
<td>0.24</td>
<td>0.31</td>
</tr>
</tbody>
</table>

however find that a reduction in the number of training samples leads to a very large performance penalty. So in general, we recommend training with more samples but fewer iterations to reduce the training time most effectively.

Finally, the hyperparameter search was only performed on a single category with a sample of the training data. This might bias the results. A hyperparameter search with the full training data would be preferable, but computationally quite expensive.

V. CONCLUSION

We present, STACC, a set of binary classifiers built on the SetFit [6] framework. Our solution is trained and tested on the NLBSE Code Comment Classification competition and greatly outperforms the average baselines in average recall, precision and F1 scores. Furthermore, STACC improves over the baseline classifiers in recall and F1 for each category. These results show that Sentence-Transformers is an efficient and performant architecture to train classifiers for code-related tasks. All of the code used to construct STACC, as well as the models themselves are publicly available.

REFERENCES


