Does BERT Understand Code? – An Exploratory Study on the Detection of Architectural Tactics in Code

Jan Keim[1][0000−0002−8899−7081], Angelika Kaplan[1], Anne Koziolek[1][0000−0002−1593−3394], and Mehdi Mirakhorli[2][0000−0003−3470−6856]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany, {jan.keim, angelika.kaplan, koziolek}@kit.edu
[2] Rochester Institute of Technology, 134 Lomb Memorial Drive, Rochester NY 14623-5608, USA, mxmvse@rit.edu

Abstract. Quality-driven design decisions are often addressed by using architectural tactics that are re-usable solution options for certain quality concerns. Creating traceability links for these tactics is useful but costly. Automating the creation of these links can help reduce costs but is challenging as simple structural analyses only yield limited results. Transfer-learning approaches using language models like BERT are a recent trend in the field of natural language processing. These approaches yield state-of-the-art results for tasks like text classification. In this paper, we experiment with treating detection of architectural tactics in code as a text classification problem. We present an approach to detect architectural tactics in code by fine-tuning BERT. A 10-fold cross-validation shows promising results with an average F_1-Score of 90%, which is on a par with state-of-the-art approaches. We additionally apply our approach on a case study, where the results of our approach show promising potential but fall behind the state-of-the-art. Therefore, we discuss our approach and look at potential reasons as well as downsides and future work.

Keywords: Software Architecture, architectural tactics, natural language processing, transfer learning, traceability, language modeling, BERT.

1 Introduction

Software traceability provides essential support for software engineering activities like coverage analysis, impact analysis, compliance verification, or testing. A problem of software traceability is the expensive creation and maintenance of traceability links [13]. Automation can reduce costs, but is challenging.

Although, the problem to detect architectural tactics is a special case of design pattern recognition, it turns out to be more challenging. Unlike design patterns that tend to be described in terms of classes and their associations [14], tactics are described in terms of roles and interactions [6]. Therefore, structural analyses only yield limited results.

Prior work by Mirakhorli et al. present an approach to detect architectural tactics in code, to trace them to requirements, and to visualize them to properly display the underlying design decision [23,24]. Their work is based on the premise that programmers use meaningful terms, e.g., for variables or methods. This is also a best practice [10] and used in other traceability approaches [3].

Recently, a lot of progress has been made in the domain of natural language processing (NLP), including text classification, by using (statistical) language models. Modern language models like the so-called *Bidirectional Encoder Representations from Transformers (BERT)* [11] can be fine-tuned on tasks such as text classification using so-called transfer learning. Hey et al. state in their introduction to BERT that fine-tuning (with BERT) for text classification is a good way to achieve good results with less training data [15]. For example, Ruder et al. [17] show that their transfer-learning approach could match performance with approaches that are trained on 100x the data. BERT and similar approaches are as of late replacing traditional discrete natural language processing pipelines [32]. However, Tenney et al. show that BERT also learns similar structures to traditional NLP pipelines.

In this work, we experiment with BERT and with the assumption that code is a special kind of text that can be used as input for BERT. Therefore, our research questions are: Do the available pretrained models of BERT understand code? Can we use language models like BERT and their transfer-learning capabilities to classify code for the detection of architectural tactics?

Thus, this paper has the following contributions: We present an approach that uses BERT to classify code that has, to the best of our knowledge, not been tried before. We evaluate our approach, compare it to others, and discuss results. Moreover, we discuss the lessons learned, especially benefits and downsides of using (natural) language models like BERT on code.

Additional details are given in our technical report [19].

2 Related Work

The most relevant related work regarding the detection of architectural tactics is by Mirakhorli et al. [23,24]. The authors use trained classifiers to detect the presence of architectural tactics like heartbeat, scheduling and authentication.

Besides that, there is other related work in the context of design pattern detection (cf. [4,9]). However, the detection of architectural tactics differs as these describe higher-level problems that can be solved using multiple different strategies.

Additional related work can be divided into three main areas: documenting design rationales, reconstructing architectural knowledge, and automated traceability. Documenting design rationales is important and different approaches try to help in this directions (cf. [5,8,25]). Unfortunately, knowledge about design decisions and architectures are mostly undocumented in many projects (cf. [16]). Therefore, researchers like Ducasse and Pollet [12] have developed techniques to reconstruct architectural knowledge. When documentation is present, approaches

that create traceability links can be used. Our approach, where we want to trace architectural design patterns, is a special case of automated trace retrieval, similar to the work by Antoniol et al. [3] and further work.

Additionally, there is some related work about the application of language models like BERT to different problems like text classification using fine-tuning.Two examples for such work are Docbert for document classification by Adhikari et al. [1] and NoRBERT for the classification of requirements [15].

Finally, approaches that are also related to this work are about building language models for code. In context of code completion and suggestion, we can find approaches that apply statistical and neural language models such as recurrent neural networks (RNNs) and N-gram (cf. [21,29]). In addition to that, further approaches also use transfer learning with code by learning on one programming language and transfer to another language, e.g., in the context of detecting code smells (cf. [30]). However, these approaches are bound to a certain application, thus not as applicable here.

3 Our Approach

We use the BERT language model fine-tuned for multi-class classification to detect architectural tactics in code. This is based on two assumptions: programmers tend to program similar functionality similarly and we can treat code like text. These assumptions are also used in other approaches (cf. [3,10]).

We train the BERT model to classify given input code into architectural tactics, including a *Unrelated* class. The inputs are classes and code snippets that should be classified for architectural tactics. Inputs are pre-processed first to omit irrelevant or not processable parts, including removal of stop-words as well as separating compound words that are written in camel case or similar. Additionally, as BERT only supports a maximum input length of 512 tokens, we truncate the input. We employ two methods for truncation: The first method is to simply truncate after the first 512 tokens; the second method removes method bodies before truncating if there are still more than 512 tokens.

We use the pre-trained uncased base model of BERT and fine-tune it. We use the the standard procedure (cf. [15]): We feed the pooled output of BERT into the classification head that consists of a single layers of linear neurons in a feedforward neural network. The softmax function gives us a probability distribution for the different outputs.

During training, we use the cross-entropy loss-function to assess the predicted distribution. Instead of a stochastic gradient descent, we use the so-called *AdamW*-optimizer [22]. AdamW usually gives better results in settings like ours.

We configure the parameters in the following way: We choose commonly used (default) parameters because of promising first empirical evidence. We use a weight decay of 0.01 and for the exponential decay rates we use a beta1 (first-moment estimates) of 0.9 for beta1 and a beta2 (second-moment estimates) of 0.999. Additionally we use a training rate of 2e-5 and a batch size of 2 to train

the classification head for our fine-tuning, based on empirical selection as well as tested parameters for text classification [31]. We perform training for ten epochs.

Our approach currently uses a multi-class, but no multi-label classifier. Therefore, we can only attach one label for each input. We do not see this as a major drawback as the case study by Mirakhorli et al. [23] shows that less than 1% of classes contain more than one architectural tactic. In the future, we plan to extend our approach to support multiple labels as well.

After fine-tuning, the trained model can be used for classification. Here, we also propose the usage of a threshold to increase the precision of our approach: If the highest confidence value of a classification is below the given threshold, the class is classified as *unrelated*.

4 Evaluation

One goal of our evaluation is to compare our results with previous results, especially the results in [23]. We are using the common evaluation metrics precision, recall, and F_1-Score to enable comparisons to the other approaches. Additionally, we reuse the data set of Mirakhorli et al. [23]. As a results, we are aiming to detect the following five architectural tactics that are represented in the available data set (cf. [23]): *Audit trail*, *Authentication*, *Heartbeat*, *Resource Pooling*, and *Scheduling*. For each of these tactics, Mirakhorli et al. identified open-source projects that implement that tactic and collected tactic-related and non-tactic-related source files. The data set consists of 50 examples for related classes and 50 examples for unrelated classes for each architectural tactic. The data sets are publicly available [26].

We first look at multiple 10-fold cross-validation experiments. We performed multiple experiments to evaluate different characteristics, all results along with our code can be found on Zenodo [18].

For the different parameter settings we can conclude the following: Increasing the amount of epochs or the batch size as well as the threshold is likely to increase precision but decrease recall. A learning rate of 2e-05 performs best in our experiments, which confirms the empirical evidence by Sun et al. [31]. We can also confirm the observation of Keskar et al. [20] that larger batches result in an inferior ability of the model to generalize. The best configuration with an F_1-Score of 90% in our case is with a learning rate of 2e-05, a batch size of two, ten epochs of training and a threshold of 0.9 during classification.

Additionally, we also observe that more data, as expected, increases the performance. However, oversampling and undersampling both do not improve results. Lastly, the two truncation methods performed similarly, with the simple truncation (F_1: 90%) slightly outperforming the method body truncation (F_1: 89%) as the recall drops when truncating method bodies.

Table 1 presents the comparison of our results with the previously reported results for the approaches (cf. [23]). Overall, our approach performs similar to others but yields relatively stable results between the different tactics, meaning the results do not vary as much between tactics compared to the other approaches.

Table 1. 10-fold cross-validation of our approach (BERT) and comparison to approaches by Mirakhorli et al. [23] using Precision (P), Recall (R), and F_1-Score. Reported F_1-Scores with asterisks do not fit to their values for precision and recall.

	SVM			Slipper			J48			Bagging			AdaBoost			Bayesian			Tactic Det.			BERT		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
Audit	.96	.46	.62	.85	.78	.81	.85	.85	.85	.88	.88	.88	.85	.85	.85	.94	.91	**.92**	.84	.92	.88	.89	.89	.89
Authentication	.91	.58	.71	.96	.94	.95	.98	.98	.92*	1.0	.92	.96	.98	.98	.94*	1.0	.80	.89	.96	.98	**.97**	.89	.87	.88
Heartbeat	.91	.62	.74	.84	.84	.84	.77	.88	.82	.89	.84	.87	.91	.86	**.89**	.92	.70	.80	.77	.92	.84	.92	.87	**.89**
Pooling	.97	.66	.79	.94	.96	.95	.94	.96	.95	.94	.94	.94	.98	.96	**.97**	.94	.96	.95	.92	.98	.95	.97	.93	.95
Scheduler	.98	.88	.93	.88	.92	.90	1.0	.98	**.99**	1.0	.98	**.99**	1.0	.98	**.99**	.96	.98	.97	.86	.88	.87	.94	.87	.90
Averages	.95	.64	.76	.89	.89	.89	.91	.93	.92	.94	.91	.93	.94	.93	.93	.95	.87	.91	.87	.94	.90	.92	.89	.90

A Friedman non-parametric statistical test indicates (disregarding the non-competitive SVM) that the difference between the results is not statistically significant. Therefore, we conclude that these classifiers perform mostly equivalently for the task of tactic detection in our 10-fold cross-validation.

We further apply our trained classifier to a case study to evaluate the performance on a large-scale project and to test how well the approach generalizes. We replicate the case study of Mirakhorli et al. [23] and detect architectural tactics in the Hadoop Distributed File System (HDFS).

Table 2. Comparative evaluation of previous approaches (cf. [23]) and our approach (BERT) for detecting architectural tactics in Hadoop using Precision (P), Recall (R), and F_1-Score.

	SVM			Slipper			J48			Bagging			AdaBoost			Bayesian			Tactic Det.			BERT		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
Audit	.08	.29	.13	.02	.29	.04	.03	.29	.06	1.0	.29	.44	.03	.29	.06	.04	.50	.07	1.0	.71	**.83**	.50	.50	.50
Authentication	.14	.52	.22	.16	.61	.26	.57	.59	.58	.58	.56	.57	.17	1.0	.30	.15	.37	.21	.61	.70	**.66**	.29	.71	.41
Heartbeat	.07	.11	.09	.31	.59	.41	.22	1.0	.36	.50	1.0	.67	.35	.96	.51	.07	.04	.05	.66	1.0	**.79**	.45	.73	.56
Pooling	.71	.11	.19	.13	.44	.20	.89	.97	**.93**	.88	1.0	**.93**	.87	.87	.87	.16	.33	.22	.88	1.0	**.93**	.89	.39	.54
Scheduler	.36	.63	.46	.65	.20	.30	.64	.87	.74	.65	.89	.75	.66	.77	.71	.32	.78	.46	.65	.94	**.77**	.62	.69	.65
Averages	.27	.33	.22	.25	.43	.24	.47	.74	.53	.72	.75	.67	.42	.78	.49	.15	.40	.20	.76	.87	.80	.55	.60	.53

The results are displayed in Table 2 and compared against the results reported by Mirakhorli et al. [23]. The promising results in the 10-fold cross-validation do not transfer to this case study and the state-of-the-art outperforms our approach. However, compared to most other approaches within the paper by Mirakhorli et al., apart from Bagging and the Tactic Detection approach, our approach still performs similar or better. In this setting, we come to the conclusion that our approach is promising, but needs further work to compete with state-of-the-art.

Although unsuccessful, we think these results provide valuable information and lessons learned. However, we think that our approach is still a valuable contribution for the community and that it is important to publish our experiences,

a view that we share with others (cf. [28]). The result demonstrates how important it is to also evaluate on different data and case studies as good cross-validation results not necessarily transfer to case studies.

5 Discussion

In this section, we want to briefly discuss our results, threats to validity, and potential future improvements to tackle the downsides of our approach.

We applied and copied commonly used experimental designs to be able to compare our approach to previous approaches as well as to mitigate potential risks to construct validity. For reproducibility, we used a randomly selected fixed (904727489) for the random number generators.

To overcome bias, we reused established data sets. This enables us comparability and increases the internal validity. However, this might affect the performance of our approach. Our data sets, both for training and for evaluation, come from the same source (cf. [23]), which causes and additional risks and is a threat to validity. The selection of training data is an important factor as well. Currently, there seems to be a problem in generalizing from the training data.

Potential issues of our approach are our assumptions that might be wrong. For example, we detect architectural tactics on a class level like previous approaches. Furthermore, we assume that we have Java code and developers use expressive, non-abbreviated variable names that are contained in BERT's dictionary.

Our approach also needs pre-processing for BERT that can influence the results (negatively). We tried to be conservative but the selection can still influence the results in various ways. However, there are some new ideas like the Longformer [7] approach that might remove input length limitations. We plan to look into them in future work.

Another risk is that BERT might look at other characteristics of the data set. Niven and Kao discovered that statistical cues in the (training) data can influence BERT's performance heavily [27]. Evaluating approaches on different case studies might help in such cases and we will look further into this.

We draw the conclusion that code is not quite the same as a common natural language text. BERT has proven to work well for text classification, but we showed that code cannot simply be treated like normal text Relations between the words in the input are different in normal text compared to code. However, BERT mainly focuses on these relations.

However, there are potential improvements to our idea of using BERT for code classification. One way is to try to transform code into a textual description in the pre-processing step with approaches like code2seq [2]. However, imprecise transformations might influence the outcome negatively (fault propagation). Another reasonable way is to adapt BERT more to our needs. We would need to train the language model on code instead of natural language texts. However, this is still an open research topic, because of differences in semantics.

We still think that transfer learning approaches are useful for tasks like the detection of architectural tactics. A clear benefit is the capability to train a

task with a rather small data set. However, the underlying approach, e.g., the language model must be suitable for the kind of input.

6 Conclusion and Future Work

In this paper, we experimented with a transfer-learning approach using the natural language model BERT to classify if classes implement certain architectural tactics. We experimented with our hypothesis that BERT can understand code similarly to text after fine-tuning. We evaluated our approach using 10-fold cross-validation with promising results. However, the approach could not compete with state-of-the-are approaches in a case study using Hadoop. Therefore, we discussed our approach further as we see a lot of potential in transfer-learning approaches.

In future work, we plan to improve our approach to perform better, e.g., by adaptations to our architecture. Additionally, we want find proper ways to either train a new language model or fine-tune one using code, so that the language model is already trained on code, which might boost the performance. We also plan to experiment with different language models beside BERT. There are reports of new language models that show better results on standard NLP tasks as well as new language models that allow longer inputs like Longformer [7].

References

1. Adhikari, A., Ram, A., Tang, R., Lin, J.: Docbert: BERT for document classification. arXiv (2019), http://arxiv.org/abs/1904.08398
2. Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code. In: ICLR (2019)
3. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE TSE **28**(10), 970–983 (Oct 2002). https://doi.org/10.1109/TSE.2002.1041053
4. Antoniol, G., Casazza, G., Di Penta, M., Fiutem, R.: Object-oriented design patterns recovery. Journal of Systems and Software **59**(2), 181–196 (2001)
5. Babar, M.A., Gorton, I.: A tool for managing software architecture knowledge. In: 2nd SHARK/ADI'07 (ICSE Workshops 2007). pp. 11–11. IEEE (2007)
6. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley Professional (2003)
7. Beltagy, I., Peters, M.E., Cohan, A.: Longformer: The long-document transformer. arXiv (2020), http://arxiv.org/abs/1904.08398
8. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: A web-based tool for managing architectural design decisions. ACM SIGSOFT **31**(5), 4 (2006)
9. Chihada, A., Jalili, S., Hasheminejad, S.M.H., Zangooei, M.H.: Source code and design conformance, design pattern detection from source code by classification approach. Applied Soft Computing **26**, 357–367 (2015)
10. Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R., Romanova, E.: Best practices for automated traceability. Computer **40**(6), 27–35 (June 2007). https://doi.org/10.1109/MC.2007.195
11. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: NAACL-HLT (2019). https://doi.org/10.18653/v1/N19-1423

12. Ducasse, S., Pollet, D.: Software architecture reconstruction: A process-oriented taxonomy. IEEE TSE **35**(4), 573–591 (2009)
13. Egyed, A., Biffl, S., Heindl, M., Grünbacher, P.: Determining the cost-quality trade-off for automated software traceability. In: 20th IEEE/ACM ASE. pp. 360–363. ACM, New York, NY, USA (2005). https://doi.org/10.1145/1101908.1101970
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Elements of reusable object-oriented software. arXiv (1995)
15. Hey, T., Keim, J., Tichy, W.F., Koziolek, A.: NoRBERT: Transfer learning for requirements classification. In: 2020 IEEE 28th RE. IEEE (2020)
16. Hoorn, J.F., Farenhorst, R., Lago, P., Van Vliet, H.: The lonesome architect. Journal of Systems and Software **84**(9), 1424–1435 (2011)
17. Howard, J., Ruder, S.: Fine-tuned language models for text classification. arXiv (2018), `http://arxiv.org/abs/1801.06146`
18. Keim, J., Kaplan, A., Koziolek, A., Mirakhorli, M.: Gram21/BERT4DAT (Jul 2020). https://doi.org/10.5281/zenodo.3925165
19. Keim, J., Kaplan, A., Koziolek, A., Mirakhorli, M.: Using BERT for the Detection of Architectural Tactics in Code. Tech. Rep. 2, Karlsruhe Institute of Technology (KIT), Karlsruhe (2020). https://doi.org/10.5445/IR/1000121031
20. Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P.: On large-batch training for deep learning: Generalization gap and sharp minima. arXiv (2016), `http://arxiv.org/abs/1609.04836`
21. Li, J., Wang, Y., Lyu, M.R., King, I.: Code completion with neural attention and pointer networks. 27th IJCAI (Jul 2018). https://doi.org/10.24963/ijcai.2018/578
22. Loshchilov, I., Hutter, F.: Fixing weight decay regularization in adam. arXiv (2017), `http://arxiv.org/abs/1711.05101`
23. Mirakhorli, M., Cleland-Huang, J.: Detecting, tracing, and monitoring architectural tactics in code. IEEE Transactions on Software Engineering **42**(3), 205–220 (March 2016). https://doi.org/10.1109/TSE.2015.2479217
24. Mirakhorli, M., Shin, Y., Cleland-Huang, J., Cinar, M.: A tactic-centric approach for automating traceability of quality concerns. In: 34th ICSE. pp. 639–649 (June 2012). https://doi.org/10.1109/ICSE.2012.6227153
25. Mirakhorli, M., Cleland-Huang, J.: Tracing architectural concerns in high assurance systems. In: 33rd ICSE. pp. 908–911. ACM (2011)
26. Mirakhorli, M., et al.: Archie. `https://github.com/SoftwareDesignLab/Archie`
27. Niven, T., Kao, H.Y.: Probing neural network comprehension of natural language arguments. In: 57th ACL (2019). https://doi.org/10.18653/v1/P19-1459
28. Prechelt, L.: Why we need an explicit forum for negative results. Journal of Universal Computer Science **3**(9), 1074–1083 (1997)
29. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: 35th ACM SIGPLAN PLDI. p. 419–428. New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594321
30. Sharma, T., Efstathiou, V., Louridas, P., Spinellis, D.: On the feasibility of transfer-learning code smells using deep learning. arXiv (2019), `http://arxiv.org/abs/1904.03031`
31. Sun, C., Qiu, X., Xu, Y., Huang, X.: How to fine-tune bert for text classification? arXiv (2019), `http://arxiv.org/abs/1905.05583`
32. Tenney, I., Das, D., Pavlick, E.: BERT Rediscovers the Classical NLP Pipeline. In: 57th ACL. pp. 4593–4601. ACL, Florence, Italy (Jul 2019). https://doi.org/10.18653/v1/P19-1452