Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2020,2
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Using BERT for the Detection of Architectural Tactics in Code

Jan Keim, Angelika Kaplan, Anne Koziolek, and Mehdi Mirakhorli

2020

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Using BERT for the Detection of Architectural Tactics in Code

Jan Keim, Angelika Kaplan, Anne Koziolek, and Mehdi Mirakhorli

{jan.keim, angelika.kaplan, koziolek}@kit.edu, 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. However, it is not sufficient to only make good design decisions but also to review the realization of design decisions in code. As manual creation of traceability links for design decisions into code is costly, some approaches perform structural analyses to recover traceability links. However, architectural tactics are high-level solutions described in terms of roles and interactions and there is a wide range of possibilities to implement each. Therefore, 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. We intent to experiment with BERT and 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 to 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 and downsides as well as potential improvements.

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

1 Introduction

Software design is a process of making decisions, therefore, design decisions are the core of every software and are essential to software development. However, these design decisions not only have to be made, but their realization and effect also needs to be checked and analyzed. 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 [9,17,21]. Automating these tasks can lower costs, but automation is still challenging, especially for tracing design decisions that are based on quality concerns such as security, reliability, performance, and safety. These quality-driven design decisions are often addressed by using architectural tactics [7,25,31]. Architectural tactics are defined as re-usable solution options for a given quality concern. Therefore, these

tactics are ways to manipulate aspects of a quality attribute through architectural design decisions [6]. For example, security concerns can be tackled with the use of the architectural tactic *authentication*. Similarly, availability of a system can be improved with the *heartbeat* tactic by monitoring critical components.

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 [19], tactics are described in terms of roles and interactions [7]. This means that a developer can use a wide range of different designs and design patterns to implement a single tactic. For example, there are at least four different ways to implement the *heartbeat* tactic including the option to not follow any specific design pattern [37]. 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 [37,38]. In their work, Mirakhorli et al. describe their *Tactic Detector* that uses information retrieval (IR) and machine learning (ML) techniques to train classifiers. The main idea is built on the tendency of programmers to use meaningful terms to name variables, methods, and classes as well as to provide meaningful comments. Using meaningful names is a best practice [14] and is also 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. Most prominently, the performance of (statistical) language models and their application has improved a lot. Modern language models like *Bidirectional Encoder Representations from Transformers (BERT)* [15] can be fine-tuned on tasks such as text classification using so-called transfer learning. It has been shown that pre-training and fine-tuning reduces the need for heavily-engineered task-specific architectures [15]. Another big advantage of fine-tuning is that less training data is needed to achieve good results. For example, Ruder et al. [24] showed that their transfer-learning approach could achieve the same performance than approaches that were trained on 100x the data.

Overall, we want to find out if BERT can "understand" code when code is used as input during fine-tuning. More precisely, we experiment with the assumption that code is a special kind of text. This allows us to use transfer-learning techniques to classify code, for example to detect architectural tactics.

This leads to our research question: Does BERT understand code? More precisely: Can we use language models like BERT and their transfer-learning capabilities to classify code for the detection of architectural tactics?

Our evaluation comes to the conclusion that our approach has potential but cannot beat the best state-of-the-art approaches. However, we think that our approach is still a valuable contribution for the community and that it is important to publish our experiences. We share the view of others (cf. [35,42,44]) that publishing negative results is important to show new research directions.

In this technical report, we provide supplementary material and further information for our study (cf. [27]).

The remainder is structured as follows: An introduction to BERT and other NLP models is given in Section 2. In Section 3, we present an extended version of related work in comparison to [27]. In more detail, we describe our approach in Section 4 and present our evaluation in Section 5. Next, we discuss our results and threats to validity. The final Section 7 concludes the technical report.

2 Introduction to Language Models and BERT

Statistical language models (LMs) aim to estimate the probabilities of sequences of words and are therefore able to predict the likelihood that a given word or sequence is following a certain sequence of words [20]. However, an important aspect of LMs like BERT [15] is their transfer learning capability. This means that LMs can be fine-tuned to a specific task other than the task they were originally trained on. The advantage of this is the little effort and training data that is needed for the fine-tuning task, as shown by Ruder et al. [24].

An important foundation of all state-of-the-art LMs are word embeddings such as *word2vec* [36]. Mikolov et al. introduce a way to calculate lower-dimensional vectors that represent words as numerical vectors based on all contexts a word appeared in the training data. However, *word2vec* only consider the syntax of a word, not its semantics, and thus disregards ambiguities. This problem is tackled by *Embeddings from Language Models (ELMo)* [43], an approach for contextualized word embeddings. The contextualization is achieved in ELMo with the usage of a bidirectional long short-term memory (LSTM) neural network with two layers in order to let the language model (LM) get a sense of the word in the context of its previous and following words. The approach *Universal Language Model Fine-tuning for Text Classification (ULMFiT)* [24] introduces additional improvements to this technique. Furthermore, ULMFiT introduces the concepts of pre-training and fine-tuning for transfer learning with LMs.

BERT by Devlin et al. [15] combines and adapts different concepts of previously released work. On release, fine-tuned BERT models outperformed state-of-the-art results on eleven NLP tasks including sequence classification, named entity recognition, and question answering [15]. To achieve this, BERT uses bidirectional pre-training similar to ELMo to incorporate context in both directions in contrast to the other unidirectional approaches, i.e., left-to-right architectures. Additionally, BERT uses an adapted version of the so-called *Transformer* architecture [49].

There are two models of BERT: the base and the large model. The base model has twelve encoder-layers, uses 768 hidden units, and twelve attention heads [15]. In comparison, the original Transformer used six encoder-layers, 512 hidden units, and eight attention heads [49]. BERT's base model has a total of 110M parameters that need to be trained. The large model of BERT uses 24 encoder-layers with encoders having 1024 hidden units, 16 attention heads and has 340M parameters in total. For both, there are additional *cased* variants that consider the capitalization of words. For training of BERT, the authors introduce *masked* language modeling [15], where about 15% of the input is masked. For

example, instead of the sentence "my dog is hairy" the input is "my dog is [MASK]." One part of the (pre-) training task is the prediction of these masked words. A second part of the training is the prediction of whether one sentence is likely to succeed another sentence. For example, the sentence "The man went to the store" is likely to be followed by "He bought a gallon of milk" but unlikely to be followed by "Penguins are flightless birds." For training, 50% of the sentences were paired with actual subsequent sentences and 50% of the time with a random sentence from the corpus. The authors choose these training tasks because the combination of encoders and their self-attention layers with bidirectionality would cause a problem in the proposed multi-layered context of BERT. In the usual language modeling task, each word would indirectly see itself and the target word and, therefore, could be trivially predicted. BERT is originally trained on the English Wikipedia and the BooksCorpus [50]. Pre-trained models of BERT are available that can directly be used for fine-tuning tasks.

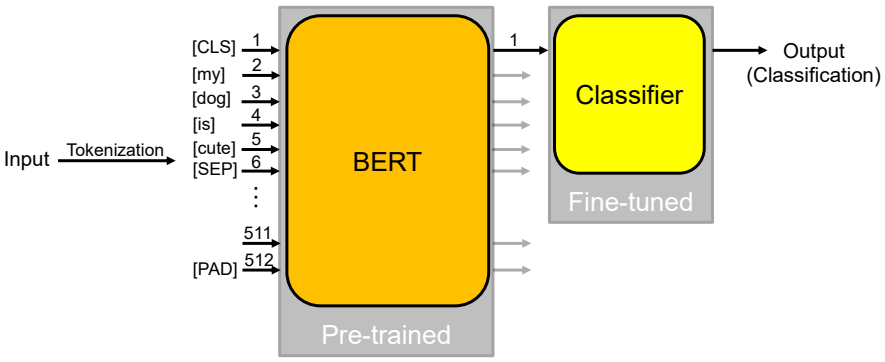

Fig. 1. Example procedure for using BERT for classification tasks (cf. [22])

BERT and similar approaches are more and more replacing traditional discrete natural language processing pipelines [48]. However, an analysis of the different layers and the underlying learned structures of BERT by Tenney et al. [48] shows that BERT remodels similar structures as the traditional NLP pipeline in its internals. An advantage of fine-tuning approaches is that they allow simpler architectures while also needing less training data. This omits the need to create heavily-engineered architectures for each task. Figure 1 shows how BERT can be used for classification. The input is tokenized and then fed into the language model. The first input token is always a special [CLS] token. For each input token, BERT outputs a (contextualized) vector. However, only the output of the first token ([CLS]), i.e., the pooled output is usually used for text classification and similar downstream tasks. This pooled output can, for example, be fed into a single-layer feedforward neural network that uses softmax to assign probabilities to different classes for classification.

3 Related Work

In this section, we complement and give an extended overview of related work with regard to [27].

The most relevant regarding the detection of architectural tactics is the work by Mirakhorli et al. [37,38]. The authors use trained classifiers to detect the presence of architectural tactics like heartbeat, scheduling, and authentication.

As the detection of architectural tactics can be seen as a special case of design pattern recognition, we can also identify several related approaches in this context of design pattern detection (cf. [4,13,18]). Although the problem seems similar, there is a difference between the problem of tactic detection and the detection of design patterns that should not be neglected: Architectural tactics describe a higher-level problem that are implemented in code, resulting in multiple different strategies to implement these tactics. Therefore, the commonly used structural analyses for detection of design patterns cannot be used and is thus more challenging.

Aside this work to detect architectural tactics or design patterns, there is related work for the detection of architectural knowledge that can be divided into the three main areas: documenting design rationales, reconstructing architectural knowledge, and automated traceability.

There are different tools and techniques for documenting design decisions and managing design decisions [5,31,39]. Some approaches even try to capture and trace architectural knowledge [12,26] and Kruchten introduced an ontology of architectural design decisions that can be used to classify design decisions [32]. However, manual labor is needed for these approaches and the cost and effort of documenting design rationales are high. Existing solutions tend to focus on the design rather than providing ways to trace the design decisions down to the code level where they are implemented [39]. As a result, developers need to manually and proactively look for information about rationale and design information.

Unfortunately, knowledge about design decisions and architectures are mostly undocumented in many projects (cf. [23]). Therefore, researchers have developed techniques to reconstruct architectural knowledge. Ducasse and Pollet collected approaches in this research direction and constructed a taxonomy [16]. The taxonomy classifies approaches into the five main axes *goals*, *processes*, *inputs*, *techniques*, and *outputs*.

When documentation is present, approaches that create traceability links can be used. Our approach is a special case of automated trace retrieval, similar to the work by Antoniol et al. [3] that perform structural analyses to recover traceability links. In our case, we want to trace architectural design patterns.

Additionally, there is some related work about the application of language models like BERT to different problems like text classification that we see closely related to our problem. One example is Docbert by Adhikari et al. [1]. The authors create state-of-the-art results for document classification by fine-tuning BERT. In a second step, the authors demonstrate that BERT can be divided into a neural model that offers high accuracy with less computation costs.

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. [33,45]). 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. [46]). However, these approaches are bound to a certain application. In the context of detecting architectural tactics we face the problem that there is only limited amount of data. This makes transfer learning appealing that learns on one task where more training data exists to another task, i.e., the detection of architectural tactics.

4 Our Approach

Our method of detecting architectural tactics in code is using the BERT language model fine-tuned for multi-class classification. Our approach is based on the two assumptions. The first assumption is that programmers tend to program similar functionality in a similar manner, thus, implement architectural tactics in similar ways. This assumption is also used by different other approaches in traceability research as is seen as best practice, i.e., for program understanding and maintenance [3,10,11,14,41]. Secondly, we see code as text, especially when we disregard execution semantics.

Our approach is divided into a training phase and the actual application. In the training phase, we use the training data to fine-tune the BERT model for classification. We train the model to classify the given input code into the given architectural tactics. We also add the class *Unrelated* that should be used if the input code is not related to one of the architectural tactics. After training, the trained classifier model can be used to classify and thus, detect architectural tactics in the input code. We describe fine-tuning and application in Section 4.2. For both steps, the input has to be pre-processed that is described in the following.

4.1 Pre-processing of input

The inputs are classes and code snippets that should be classified for architectural tactics. This input needs to be pre-processed first to remove some irrelevant parts and parts that cannot be processed by BERT.

One issue with BERT is its limited input length. Currently, the released pre-trained BERT models are only able to process input with a maximum length of 512 tokens. Classes often exceed this limit of 512 tokens, therefore, the input needs to be truncated in a second pre-processing step. For this, we employ two methods: The first method is to simply truncate after the first 512 tokens; the second method removes method bodies. If the output is still too long, we simply truncate it after 512 tokens as in the first method.

Additionally, we apply further pre-processing beforehand: Removal of stop-words and separating compound words. In text classification it is common to

remove stop words because they are common words that usually provide only little value for processing text, especially when using statistical approaches. These stop words include articles, possessive nouns, and pronouns like "the", "a", "his", "hers", "he", or "she". Additionally, we remove specific keywords of the target programming language. Besides stop words, we ignore lines containing only information about licensing that can usually be found at the beginning of a file.

In code, identifiers that represent compound nouns are usually written in some special kind: In camel case using capital letters to indicate a new word (like in "camelCase"), in snake case using underscores to separate words (like in "snake_case"), or in kebab-case using hyphens (like in "kebab-case"). For better processing and compatibility with BERT, we split up the identifiers that represent compound nouns. Lastly, we remove symbols that BERT cannot deal with, including semicolons, equal signs and (curly) brackets.

4.2 Fine-tuning BERT for detection of architectural tactics

For fine-tuning BERT for our approach, we use the pre-trained uncased base model of BERT. The base model uses far less computational resources compared to the large model. We use the the standard procedure (cf. Section 2): A model with a classification head on top, i.e., a linear layer on top of the pooled output which is the output of first token, i.e., the [CLS] token of the input. 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. The classification head outputs a probability for each label-class in our classification.

During training, we use the cross-entropy loss-function. This means that we try to quantify how close the predicted distribution is to the true distribution. The cross-entropy loss-function punishes wrong or uncertain predictions and analogously rewards confident predictions that are correct. This is captured in the following formula:

$$H(p, q) = -\sum_{x} p(x) \log q(x) \tag{1}$$

In this formula, x denotes the different labels, $p(x)$ stands for the target probability, and $q(x)$ the actual probability for the label x. In our case, $p(x)$ equals 1.0 for a correct label and $p(x)$ equals 0.0 for the incorrect labels.

An important component of (deep) neural networks is the optimizer that updates the various weights within a network. Usually, weights are updated in the network using a classical stochastic gradient descent. However, we use the so-called $AdamW$-optimizer [34] that is an adaptation of the popular $Adam$-optimizer [30]. AdamW implements a weight decay correction and does not compensate for bias as in the regular Adam-optimizer and usually gives better results in settings like ours. As optimizer setting, we do not use a warm-up phase to increase the performance on sparse datasets.

We configure the parameters in the following way: We choose commonly used (default) parameters because of promising first empirical evidence (cf. Section 5.2).

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 [47]. With these values, we train the classifier 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. [37] 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. However, classes implementing architectural tactics are less common than classes unrelated to architectural tactics. For example, the analysis of *Apache Hadoop* for ten architectural tactics in [37] shows that only 9.42% of classes are related to at least one of the ten covered architectural tactics. Therefore, 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*.

5 Evaluation

The evaluation of our approach is split into different parts. We first describe the used (training) data in Section 5.1. Afterwards, we evaluate our approach and compare it to state-of-the-art approaches. For this, we perform a 10-fold cross-validation on the training data in Section 5.2. Furthermore, we evaluate the performance of our approach on a case study and again compare the performance of our approach to other approaches in Section 5.3. In these evaluations, we are using the common evaluation metrics precision, recall, and F_1-Score to enable comparisons to the other approaches.

5.1 Training Set

One goal of our evaluation is to compare our results with previous results, especially the results in [37]. Therefore, we reused the data set of Mirakhorli et al. [37] as well as one older data set by Mirakhorli et al. [38]. As a results, we are aiming to detect the following five architectural tactics (cf. [37,38]): *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 first data set by Mirakhorli et al. [38] consists of eleven examples for related classes per architectural tactics (55 total) and 220 examples for unrelated classes in total. The second, bigger data set by Mirakhorli et al. [37] consists of 50 examples for related classes and 50 examples for unrelated classes for each architectural tactic. The data sets are publicly available in an open-source repository (cf. [28]).

5.2 10-fold cross-validation

For evaluation, we first look at multiple 10-fold cross-validation experiments. This means, we evaluate ten times using each time (different) 10% of the data for evaluation and train on the other 90%. We performed multiple experiments to evaluate different characteristics.

Table 1. Average precision (Prec.), recall (Rec.), and F_1-Score for different configurations including different amount of epochs (ep.), different confidence thresholds (thr.), various batch sizes (bs.), and learning rates (lr.) of our approach.

Configuration	$\overline{\text{Prec.}}$	$\overline{\text{Rec.}}$	$\overline{F_1}$
10 ep., thr. 0.9, lr. 2e-05, bs. 2	0.92	0.89	0.90
20 ep., thr. 0.9, lr. 2e-05, bs. 2	0.92	0.87	0.89
10 ep., **thr. 0.9**, lr. 2e-05, bs. 2	0.92	0.89	0.90
10 ep., **thr. 0.5**, lr. 2e-05, bs. 2	0.88	0.88	0.88
10 ep., **thr. 0.0**, lr. 2e-05, bs. 2	0.88	0.88	0.88
10 ep., thr. 0.9, **lr. 2e-05**, bs. 2	0.92	0.89	0.90
10 ep., thr. 0.9, **lr. 3e-05**, bs. 2	0.89	0.87	0.88
10 ep., thr. 0.9, lr. 2e-05, **bs. 2**	0.92	0.89	0.90
10 ep., thr. 0.9, lr. 2e-05, **bs. 8**	0.93	0.85	0.89

As first experiment, we evaluate the different configurations of our approach in a 10-fold cross-validation with a fixed seed (904727489) for reproducibility using the bigger data set. We compare different learning rates, amount of epochs, as well as different confidence thresholds. We select common values for these settings and aim to confirm our intuitions: Increasing the amount of epochs or the batch size is likely to increase precision but decrease recall. Moreover, higher thresholds increases the precision but may decrease the recall. In Table 1, the average results for a selection of the different tested configurations are displayed. We can confirm our intuition that increasing the amount of epochs increases the precision but decreases the recall. More epochs causes the classifier to fit better to the training data. However, an over-fitting probably takes place, causing the decrease of recall. Similar results can be seen for the threshold value. A higher threshold means the classifier needs a high confidence to label a class. As expected, lowering the threshold decreases the precision and overall worsens the results. The learning rate of 2e-05 performs best in our experiments. This confirms the empirical evidence by Sun et al. [47] that showed best results for text classification using this learning rate. Lastly, we look at different batch sizes. The batch size determines the number of training samples that are processed before updating the network, i.e., the weights. According to Keskar et al. [29], larger batches result in lower quality of the model in regard to its ability to generalize. Here, we can observe that increasing the batch size reduces the recall, but slightly increases the precision. However, increasing the batch size also increases the memory consumption on the GPU, so there are (hardware) limits for this parameter. The

best configuration in our case can be derived with a learning rate of 2e-05, a batch size of two, ten epochs of training and a threshold of 0.9 during classification.

Table 2. Results for 10-fold cross-validation for different training data, including Precision (Prec.), Recall (Rec.), and F_1-Score

	data set from [38]			data set from [37]		
	Prec.	Rec.	F1	Prec.	Rec.	F1
Audit	0.60	0.70	0.65	0.89	0.89	0.89
Authentic.	0.57	0.60	0.58	0.89	0.87	0.88
Heartbeat	0.40	0.38	0.39	0.92	0.87	0.89
Pooling	0.30	0.25	0.27	0.97	0.93	0.95
Scheduling	0.25	0.30	0.27	0.94	0.87	0.90
Averages	0.42	0.45	0.43	0.92	0.89	0.90

Besides evaluating different parameters, we evaluate how different data sets are influencing the performance. The results for the different data sets in Table 2 clearly show that the larger data set outperforms the smaller data set in this setting. The average precision and recall more than doubles for the second data set compared to the first. This comes to no surprise as larger data sets generally perform better in deep learning approaches. We also evaluated the two truncation methods that we used. Overall, both truncation methods are pretty close to each other. The F_1-Score for simple truncation is at 90% (Precision 92%, Recall 89%) while the F_1-Score for truncating method bodies is at 88% (Precision 92%, Recall 86%). In this scenario, precision is equal and recall for the truncation of method bodies is slightly worse.

Table 3. 10-fold cross-validation of our approach (BERT) and comparison to approaches by Mirakhorli et al. [37] using precision (P), recall (R), and F_1-Score. (Previously reported) F_1-Scores with asterisks do not fit to their corresponding 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

Table 3 presents the comparison of our results with the previously reported results for the approaches (cf. [37]). As we use the same data set as Mirakhorli et al. [37], we can directly compare our results to the reported results. Our approach

performs similar to other approaches. Overall, our approach yields relatively stable results between the different tactics, meaning the results do not vary as much between tactics compared to the other approaches.

We applied the Friedman non-parametric statistical test to figure out if the results are significantly different. We disregard the clearly non-competitive SVM approach for this test. The test results indicate that the difference between the results is not statistically significant. Following this result, we conclude that these classifiers perform mostly equivalently for the task of tactic detection when considering 10-fold cross-validation. We also evaluated using undersampling and oversampling, i.e., providing equal share of training data by randomly removing or adding data until classes are equally sized. We come to the conclusion that these techniques do not improve the results in our settings.

5.3 Case study

The main purposes of our approach is the detection of architectural tactics in large-scale projects. Therefore, we apply our trained classifier to a case study to evaluate the performance on a large-scale project. Additionally, we can test how well the approach generalizes from the training data. We chose to replicate the case study of Mirakhorli et al. [37], i.e., we aim to detect architectural tactics in the Hadoop Distributed File System (HDFS)[1].

Table 4. Comparative evaluation of previous approaches and of 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 4 and compared against the results reported by Mirakhorli et al. [37]. Our approach yields promising results in the previous 10-fold cross-validation, however the results do not transfer to this case study. The Tactic Detection approach by Mirakhorli et al. [37] 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 conclude that our approach is still promising, but needs further work to compete with state-of-the-art.

[1] See also `https://hadoop.apache.org/docs/stable/`

Nevertheless, we think these results provide valuable information and lessons learned. They demonstrate how important it is to also evaluate on different data and case studies (for generalizability of an approach) as good cross-validation results not necessarily transfer to case studies. Although transfer-learning approaches promise good generalization this does not seem to be the case here, which is another key aspect to take away. Further discussions about our approach can be found in Section 6.

6 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 seed for the random number generators.

To overcome bias, we did not create an own data set but reused established data. On one hand, this enables us comparability and increases the internal validity. On the other hand, however, this might affect the performance of our approach. We believe that adding additional and more diverse data to the training set will most likely increase the performance. Currently, there seems to be a problem in generalizing from the training data. This can be seen in the different performance for 10-fold cross-validation in comparison to the evaluation on the case study. Similarly, when training on the smaller data set compared to training on the bigger data set, we can see noticeable differences in the performance. Therefore, applying our approach using a better and bigger training set is one of the things we plan to do in future work.

Potential issues of our approach are the assumptions we made that might be wrong. For example, we tried to detect architectural tactics on a class level like previous approaches. However, some architectural tactics might need other scopes. Furthermore, we assumed that the code is in Java and the developers use expressive, non-abbreviated variable names. Abbreviations and non-expressive terms are unlikely to be contained in BERT's dictionary, therefore can also influence the result negatively.

To use BERT, our approach also needs pre-processing like the removal of certain features and characters. The pre-processing, though, can influence the results (negatively). We tried to be as conservative as possible but the selection still might skew the results in various ways. For example, truncating classes can potentially remove parts that are relevant for a specific tactic. However, there are some new ideas like the Longformer approach [8] that might remove this limitation as it promises longer input lengths with similar results.

It is also possible that BERT is not particularly looking at the text itself but at other characteristics. Niven and Kao discovered that statistical cues in the (training) data can influence BERT's performance heavily [40]. Evaluating

approaches on different and varying case studies might help in such cases and we will look further in this potential threat.

We can 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 this does not translate properly to code. This shows that code cannot simply be treated like normal text. We have to look more detailed into how BERT works: BERT contextualizes the input and is trained on normal text to "understand" the relations between the words in the input. These relations are different in normal text compared to code.

However, there are potential improvements to our idea of using BERT for code classification. One way is to experiment with approaches like code2seq [2] that transform code into a textual description in the pre-processing step. A negative side to these approaches might be the chance that needed information in the code gets obscured. Additionally, faults in these approaches will most likely influence the outcome negatively (fault propagation). Another reasonable way is to adapt BERT to our needs. We would need to adapt language models for code and train them on code instead of natural language texts. However, adapting language models for code is not trivial and still an open research topic. This is mostly because code semantics, especially underlying execution semantics, are hard to capture compared to semantics in text.

Overall, although our approach has limitations and does not bring state-of-the-art results, we still think the results are valuable and some lessons can be learned. The results are some first experiences of applying a language model like BERT to code and bring forward the issues that such an approach brings. 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.

7 Conclusion

In this technical report, we gave supplementary material and more details for the study in [27]. Based on our hypothesis that BERT can understand code similarly to text after fine-tuning, we experimented with a transfer-learning approach using such a natural language model to classify if classes implement certain architectural tactics. We evaluated our approach using 10-fold cross-validation with promising results. The approach could not compete with state-of-the-art approaches in a case study using Hadoop. Nevertheless, we still think that these results and experiences are important. Therefore, we discussed our approach in more detail as we see a lot of potential in transfer-learning approaches for further research.

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: International Conference on Learning Representations (2019), https://openreview.net/forum?id=H1gKYo09tX
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. Bachmann, F., Bass, L., Klein, M.: Deriving architectural tactics: A step toward methodical architectural design. Tech. rep., Carnegie-Mellon Uni (2003)
7. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley Professional (2003)
8. Beltagy, I., Peters, M.E., Cohan, A.: Longformer: The long-document transformer. arXiv:2004.05150 (2020)
9. Berenbach, B., Gruseman, D., Cleland-Huang, J.: Application of just in time tracing to regulatory codes. In: Proceedings of the CSER (2010)
10. Biggerstaff, T.J.: Design recovery for maintenance and reuse. Computer **22**(7), 36–49 (1989)
11. Biggerstaff, T.J., Mitbander, B.G., Webster, D.: The concept assignment problem in program understanding. In: Proceedings of WCRE. pp. 27–43. IEEE (1993)
12. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: A web-based tool for managing architectural design decisions. ACM SIGSOFT **31**(5), 4 (2006)
13. 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)
14. 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
15. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Proceedings of the 2019 NAACL-HLT. pp. 4171–4186. ACL, Minneapolis, Minnesota (Jun 2019). https://doi.org/10.18653/v1/N19-1423
16. Ducasse, S., Pollet, D.: Software architecture reconstruction: A process-oriented taxonomy. IEEE TSE **35**(4), 573–591 (2009)
17. Egyed, A., Biffl, S., Heindl, M., Grünbacher, P.: Determining the cost-quality trade-off for automated software traceability. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 360–363. ASE '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1101908.1101970
18. Fontana, F.A., Zanoni, M., Maggioni, S.: Using design pattern clues to improve the precision of design pattern detection tools. JOT **10**(4), 1–31 (2011)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Elements of reusable object-oriented software. arXiv (1995)
20. Goldberg, Y.: Neural network methods for natural language processing. Synthesis Lectures on Human Language Technologies **10**(1), 1–309 (2017)
21. Gotel, O., Finkelstein, A.: Extended requirements traceability: results of an industrial case study. In: Proceedings of ISRE '97. pp. 169–178 (Jan 1997). https://doi.org/10.1109/ISRE.1997.566866
22. Hey, T., Keim, J., Tichy, W.F., Koziolek, A.: Norbert: Transfer learning for requirements classification. In: 2020 IEEE 28th RE. IEEE (2020)

23. Hoorn, J.F., Farenhorst, R., Lago, P., Van Vliet, H.: The lonesome architect. Journal of Systems and Software **84**(9), 1424–1435 (2011)
24. Howard, J., Ruder, S.: Fine-tuned language models for text classification. arXiv (2018), http://arxiv.org/abs/1801.06146
25. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: WICSA 2005. pp. 109–120 (Nov 2005). https://doi.org/10.1109/WICSA.2005.61
26. Jansen, A., Van Der Ven, J., Avgeriou, P., Hammer, D.K.: Tool support for architectural decisions. In: WICSA 2007. pp. 4–4. IEEE (2007)
27. Keim, J., Kaplan, A., Koziolek, A., Mirakhorli, M.: Does BERT understand code? – an exploratory study on the detection of architectural tactics in code. In: 14th European Conference on Software Architecture (ECSA 2020). Springer International Publishing (2020)
28. Keim, J., Kaplan, A., Koziolek, A., Mirakhorli, M.: Gram21/BERT4DAT (Jul 2020). https://doi.org/10.5281/zenodo.3925165
29. 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
30. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv (2014), http://arxiv.org/abs/1412.6980
31. Kruchten, P., Capilla, R., Dueñas, J.C.: The decision view's role in software architecture practice. IEEE Software **26**(2), 36–42 (March 2009). https://doi.org/10.1109/MS.2009.52
32. Kruchten, P.: An ontology of architectural design decisions in software intensive systems. In: 2nd Groningen workshop on software variability. pp. 54–61 (2004)
33. Li, J., Wang, Y., Lyu, M.R., King, I.: Code completion with neural attention and pointer networks. Proceedings of the Twenty-Seventh IJCAI (Jul 2018). https://doi.org/10.24963/ijcai.2018/578
34. Loshchilov, I., Hutter, F.: Fixing weight decay regularization in adam. arXiv (2017), http://arxiv.org/abs/1711.05101
35. Matosin, N., Frank, E., Engel, M., Lum, J.S., Newell, K.A.: Negativity towards negative results: a discussion of the disconnect between scientific worth and scientific culture. Disease Models & Mechanisms **7**(2), 171–173 (2014). https://doi.org/10.1242/dmm.015123
36. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient Estimation of Word Representations in Vector Space. arXiv (Jan 2013), https://arxiv.org/abs/1301.3781
37. 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
38. Mirakhorli, M., Shin, Y., Cleland-Huang, J., Cinar, M.: A tactic-centric approach for automating traceability of quality concerns. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 639–649 (June 2012). https://doi.org/10.1109/ICSE.2012.6227153
39. Mirakhorli, M., Cleland-Huang, J.: Tracing architectural concerns in high assurance systems. In: Proceedings of the 33rd ICSE. pp. 908–911. ACM (2011)
40. Niven, T., Kao, H.Y.: Probing neural network comprehension of natural language arguments. In: Proceedings of the 57th ACL. pp. 4658–4664 (Jul 2019). https://doi.org/10.18653/v1/P19-1459
41. Oman, P.W., Cook, C.R.: The book paradigm for improved maintenance. IEEE Software **7**(1), 39–45 (1990)

42. Paige, R.F., Cabot, J., Ernst, N.A.: Foreword to the special section on negative results in software engineering. Empirical Software Engineering **22**(5), 2453–2456 (2017). https://doi.org/10.1007/s10664-017-9498-0
43. Peters, M.E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., Zettlemoyer, L.: Deep contextualized word representations. In: Proc. of NAACL (2018)
44. Prechelt, L.: Why we need an explicit forum for negative results. Journal of Universal Computer Science **3**(9), 1074–1083 (1997)
45. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN PLDI. p. 419–428. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594321
46. 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`
47. Sun, C., Qiu, X., Xu, Y., Huang, X.: How to fine-tune bert for text classification? arXiv (2019), `http://arxiv.org/abs/1905.05583`
48. Tenney, I., Das, D., Pavlick, E.: BERT Rediscovers the Classical NLP Pipeline. In: Proceedings of the 57th ACL. pp. 4593–4601. ACL, Florence, Italy (Jul 2019). https://doi.org/10.18653/v1/P19-1452
49. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. pp. 6000–6010. NIPS'17, Curran Associates Inc., Long Beach, California, USA (Dec 2017)
50. Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., Fidler, S.: Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books. In: 2015 IEEE ICCV. pp. 19–27 (Dec 2015). https://doi.org/10.1109/ICCV.2015.11