Making the Most of Small Software Engineering Datasets With Modern Machine Learning

Julian Aron Prenner and Romain Robbes

Abstract—This paper provides a starting point for Software Engineering (SE) researchers and practitioners faced with the problem of training machine learning models on small datasets. Due to the high costs associated with labeling data, in Software Engineering, there exist many small ($<$ 5,000 samples) and medium-sized ($<$100,000 samples) datasets. While deep learning has set the state of the art in many machine learning tasks, it is only recently that it has proven effective on small-sized datasets, primarily thanks to pre-training, a semi-supervised learning technique that leverages abundant unlabelled data alongside scarce labelled data. In this work, we evaluate pre-trained Transformer models on a selection of 13 smaller datasets from the SE literature, covering both, source code and natural language. Our results suggest that pre-trained Transformers are competitive and in some cases superior to previous models, especially for tasks involving natural language; whereas for source code tasks, in particular for very small datasets, traditional machine learning methods often has the edge. In addition, we experiment with several techniques that ought to aid training on small datasets, including active learning, data augmentation, soft labels, self-training and intermediate-task fine-tuning, and issue recommendations on when they are effective. We also release all the data, scripts, and most importantly pre-trained models for the community to reuse on their own datasets.

Index Terms—Small datasets, transformer, BERT, RoBERTA, pre-training, fine-tuning, data augmentation, back translation, soft labels, active learning

1 Introduction

Small datasets are commonplace for many Software Engineering problems. While the creation of a labelled dataset is always a significant undertaking, this is even more the case for Software Engineering. In many cases, significant expert knowledge is required to label Software Engineering data, making it difficult to use crowd-sourcing techniques, as is often done in other fields such as in computer vision [1]. Moreover, some labelling tasks involve detailed (text or source code) understanding, making the labelling of a single example time consuming. Dataset size may be further reduced by the need to label the same examples multiple times and to compute inter-rater agreement. Due to all these factors, it is thus not uncommon for hand-labelled Software Engineering datasets to number only a few thousands or even hundreds of samples. For instance, the 13 datasets used in this work (described in Section 2) range from 200 to 62,275 samples, with three datasets having more than 5,000 samples, and four having less than 1,000. In this work, we consider datasets with less than 5,000 samples "small"; those with less than 100,000 as "medium sized".

Historically, small Software Engineering datasets were used with traditional machine learning algorithms, such as Support Vector Machines (SVMs), Logistic Regression or Random Forests, often combined with manual feature engineering. In recent years, early experiments with deep learning architectures [2], [3], [4], [5], such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), showed mixed results, suggesting that for many tasks where training data is scarce, deep learning does not provide a clear benefit, especially in light of its considerably higher computational costs.

Whether deep learning is in fact not well suited for these small datasets, and if so, for which kind of tasks and dataset sizes is the central question of this paper. The motivation to take a second look at this problem is the recent advent of semi-supervised learning [6]. Semi-supervised learning is a machine learning paradigm in which both labelled data and unlabelled data are leveraged in the learning process, with the latter being much cheaper to acquire. While prevalent in computer vision, it is only since 2018 that semi-supervised learning has become viable in the NLP domain [7], [8], in the form of pre-training. Since 2019, pre-trained Transformer-based models such as BERT [9] or RoBERTa [10] have set many records in NLP and related fields, and considerably improved the state of the art on important benchmarks such as GLUE [11] and SQuAD [12], in which there are small datasets. In addition to pre-training, several additional techniques have the potential to benefit small datasets, including Domain-specific Fine-Tuning, Intermediate-Task fine-tuning, Active Learning, Self-Training, Data Augmentation, and Soft Labels. Thus, a better understanding of when to combine these techniques is necessary. We provide background on

• *The authors are with the Faculty of Computer Science, Free University of Bozen-Bolzano, 39100 Bolzano, Italy. E-mail: prenner@inf.unibz.it, rrobbes@unibz.it.*

Manuscript received 21 June 2021; revised 29 Oct. 2021; accepted 4 Dec. 2021. Date of publication 16 Dec. 2021; date of current version 12 Dec. 2022.
This work was supported in part by IDEALS and ADVERB Projects from the Free University of Bozen-Bolzano. Parts of the results of this work were computed on the Vienna Scientific Cluster (VSC).
(Corresponding author: Julian Aron Prenner.)
Recommended for acceptance by D. Lo.
Digital Object Identifier no. 10.1109/TSE.2021.3135465

the Transformer Architecture, Semi-supervised learning via pre-training and other techniques in Section 3.

While previous work showed promising results in applying pre-training to SE problems [13], [14], [15], [16], [17], this work examines this phenomenon in more depth, by applying the pre-training paradigm on thirteen different small and medium-sized Software Engineering datasets selected from the literature. These datasets span natural language, source code, and source code comments, in a variety of domains (several sentiment analysis tasks, several app review classification tasks, technical debt detection, comment classification, code comment coherence, code smell detection, code readability, code complexity). In addition, and unlike previous work, we also investigate the impact of the additional techniques mentioned above, when they are relevant. Section 4 presents methodological details such as pre-processing, baselines, and training, testing and validation modalities, for all the scenarios we consider. This section also presents the pre-trained and fine-tuned models we use in this work, including StackOBERTflow, a Transformer model pre-trained on 26 million Stack Overflow comments.

Section 5 presents the results of the paper, answering the following research questions:

RQ1. For which domains and tasks does the pre-training paradigm outperform the baselines and which pre-training regimen is most effective? We find that pre-training is effective for tasks working on natural language and source code comments, but is not as effective for tasks working on source code yet; further pre-training is the most effective strategy in most cases.

RQ2. Which additional techniques are effective, and if so in which circumstances? We find that some techniques, such as domain-specific pre-training, and data augmentation are effective in some (but not all) settings, while we find limited evidence for the effectiveness of others, such as active learning.

Finally, we close the paper by documenting the limitations of our study, and the opportunities for additional studies in Section 6.2. We conclude the work in Section 7, summarizing initial recommendations on the effectiveness of pre-training and the additional techniques. Additional material can be found in three appendices: Appendix A provides additional information on datasets; Appendix B provides results; last but not least, Appendix C provides instructions on how to access the data, scripts, and pre-trained models we used in our experiments, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2021.3135465. These models can be fine-tuned for a wide range of tasks relating to software artifacts; we hope that they will prove useful to other researchers in the field.

2 Datasets and Related Work

We selected thirteen datasets introduced in the Software Engineering literature in recent years, aiming for both variety in terms of artifacts, dataset size, and classification tasks [2], [3], [18], [19], [20], [21], [22], [23], [24], [25], [26]. Another selection factor was the availability of a comparable baseline or a way to reproduce the initial experiment. Seven datasets involve natural language, two code comments, and the remaining four source code (one with comments). They vary from 341 to 62,275 examples, and from 2 to 16 classes. The datasets cover nine different tasks: a) sentiment classification of software artifacts, such as Stack Overflow comments, app and code reviews b) detection of informative app reviews c) classification of app reviews d) detection of self-admitted technical debt through code comments e) classification of code comments f) prediction of code-comment coherence g) detection of linguistic code smells h) prediction of code runtime complexity and i) prediction of code readability.

TABLE 1
Datasets Considered in This Work, Along With Their Size, Number of Classes and Usage

Name	Size	# Cl.	Type	Usage
Sentiment Classification (Stack Overflow) [18]	4,423	3	Natural language	Train, Test, Valid.
Sentiment Classification (Stack Overflow) [19]	1,500	3	Natural language	Test
Sentiment Classification (JIRA Issues) [3]	926	2	Natural language	Test
Sentiment Classification (App Reviews) [19]	341	3	Natural language	Test
Informative App Review Detection [20]	12,000	2	Natural language	Train, Test, Valid.
App Review Classification [21]	3,691	4	Natural language	Train, Valid.
App Review Classification [34]	3,000	7	Natural language	Train, Valid.
Self-Admitted Technical Debt Detection [22]	62,275	2	Comments	Train, Test, Valid.
Comment Classification [23]	11,232	16	Comments	Train, Valid.
Code-Comment Coherence Prediction [24]	2,881	2	Code w/ Comments	Train, Test, Valid.
Linguistic Smell Detection [2]	1,753	2	Code	Train, Valid.
Code Runtime Complexity Classification [25]	933	5	Code	Train, Valid.
Code Readability Prediction [26]	200	2	Code	Train, Valid.

Next, we discuss general related work as well as related work to each of these tasks. For a more concise overview refer to Table 1.

Related Work. Wang *et al.* [27] analyzed code-comment coherence by means of a Bi-LSTM model which was evaluated also on the above dataset. However, because a different evaluation methodology was used, a direct comparison was not possible. Arnaoudova *et al.* [28], in addition to introducing the already mentioned code smell taxonomy, provide an exhaustive treatment of this subject, including also an in-depth empirical study of how developers perceive such smells. A system to detect linguistic code smells in infrastructure as code scripts was developed by Borovits *et al.* [29]. There exists further literature on smell detection in a broader context: for instance, Fontana *et al.*[30] presented a machine learning approach to code smell detection whose results, however, were called into question in a later

replication study [31]. More recently, Sharma *et al.* [32] used deep learning models such as CNNs and LSTM networks to detect code smells in C# and Java code. Also related, is work done by Arcelli Fontana and Zanoni [33], who experiment with machine learning models for code smell *severity* prediction, which can be considered an extension of the simpler detection task.

2.1 Natural Language Datasets

These datasets contain mainly natural language, but may occasionally contain some source code identifiers. They are the closest to the original setting for models pre-trained on a generic English corpus, although they come from very specific domains.

Sentiment Classification. In sentiment classification, a model assigns a sentiment class (e.g., one of *positive*, *negative*, *neutral*) to a sentence or short piece of text; in our case the text's domain is related to software development.

The dataset compiled by Calefato *et al.* (Senti4SD [18]) contains 3,097 training and 1,326 test samples each labeled as either *positive*, *negative* or *neutral*; all samples were extracted from questions, answers and comments of Stack Overflow posts. Along with their dataset, the authors also released individual rater annotations, i.e., three rater labels per sample, from which the final labels were obtained by applying a majority vote rule. The dataset is used to evaluate an SVM classifier using word embeddings trained on a Stack Overflow corpus as features; we include this SVM as a baseline (see Table 3).

Lin *et al.* [3] created a sentiment classification datasets consisting of 1,500 sentences (from 178 text fragments) extracted from Stack Overflow discussions. They also adapted two previous datasets of 636 Jira issues (926 sentences), and 130 app reviews (341 sentences) [35], [36]. The JIRA issues dataset has only two sentiment classes (*positive* and *negative*) while the other two have an additional *neutral* class. All three datasets were used in a study of several sentiment analysis tools, and a novel model introduced in the same work [3]; we include all of them as baselines in our comparison (Table 3). To allow for a comparison with these baselines, all three datasets are *only used as test sets* in this work.

Informative App Review Detection. An app review is considered informative if it contains valuable information for the application developer, such as feature suggestions or bug reports. The dataset presented by Chen *et al.* [20] contains 12,000 app reviews belonging to four popular mobile apps from the Google Play Store. The dataset is partitioned into predefined test (2000 samples per app) and train sets (1000 samples per app). Three raters annotated each review as either *informative* or *non-informative*, with a majority vote to determine the final label. This work also presented AR-Miner, a tool based on an expectation maximization with Naive Bayes (EMNB) classifier to detect such informative app reviews; this tool is our baseline.

App Review Classification. Maalej *et al.* [21] compiled a dataset of 4,400 reviews crawled from Apple's App Store and Google Play. Each review was categorized by two raters into one of four classes (*bug report*, *feature request*, *user experience* or *rating*); reviews with rater disagreement were discarded. The authors experimented with various types of classifiers, finding that an ensemble of binary classifiers performs considerably better than a single multiclass classifier. We only study multiclass classification, for comparability reasons, use the author's multiclass classifier as a baseline. Scalabrino *et al.* [34] and Villarroel *et al.* [36] introduce CLAP, a tool for automatic classification and clustering of app reviews that is evaluated on a dataset of 3,000 app reviews created by the same authors. We use the CLAP dataset in a data augmentation experiment.

Related Work. There is a vast literature on sentiment analysis and classification. For a general overview see e.g., Mäntylä *et al.* [37]. Zhang *et al.* [15] provide an in-depth comparison of pre-trained Transformers with six different sentiment analysis tools, including both, tools for general sentiment analysis and tools specifically targeted towards Software Engineering. The study uses some datasets that we also use (e.g., from Lin *et al.* [3]), they defined custom training and testing sets while we used them solely for testing, which makes comparisons difficult. While they explored additional Transformer architectures (XLNet [38] and ALBERT [39]), their study was limited to only fine-tuning: they did not investigate the use of task-specific pre-training, nor any other of the additional methods that we study. Ahmed *et al.* [40] introduced SentiCR, a sentiment analysis tool that uses of Part Of Speech (POS) tags and Gradient Boosting Trees, while Chen *et al.* [41] present SEntiMoji, a model that leverages emojis to improve SE sentiment classification.

Dhinakaran *et al.* [42] investigated the application of *active learning* for app review analysis, using the dataset by Maalej *et al.* They employ traditional machine learning algorithms (naive Bayes, logistic regression, and SVM). A similar experiment, carried out on the same dataset can also be found in this work.

2.2 Datasets of Code Comments

Source code comments somewhat differ from natural language: they may often contain source code identifiers, code annotations, and specific idioms common in source code documentation.

Self-Admitted Technical Debt Detection. Self-admitted technical debt (SATD) is technical debt known to and acknowledged by the author. It is often expressed in code comments with a short description of a flaw or shortcoming and sometimes, but not always, marked with specific keywords, such as `FIXME`, `TODO` or `HACK`. Detection of such comments can be useful to assess software quality, aid decision-making or direct further development.

The dataset by S. Maldonado *et al.* [22] contains SATD comments extracted from 10 prominent Java projects (for more details, see Table 13 in Appendix A, available in the online supplemental material). With over 60,000 samples, it is by far the largest dataset used in this work. Each dataset sample is assigned to one of five SATD categories informed by an established ontology of technical debt [43] (*design debt*, *requirement debt*, *defect debt*, *documentation debt* or *test debt*) or labeled as not containing any SATD at all. In this work, we concentrate on the binary version of the problem (detecting presence of SATD), for two reasons: (1) because there are large class imbalances (i.e., the document debt class makes up less than 0.1% of the total data), and (2) to

compare performance to binary classifiers from previous work. In addition to introducing this dataset, the authors also perform various SATD detection and classification experiments using traditional NLP and machine learning methods; we use their SATD detection model as a baseline. Baselines also include the CNN-based approach from Ren *et al.* [4] as well as the more recent HATD system by Wang *et al.* [44], which combines self-attention with ELMo [45], word embeddings extracted from a pre-trained LSTM Language Model.

Comment Classification. Pascarella and Bacchelli [23] released a dataset of over 11,000 comments from open-source Java projects classified according to a taxonomy of 16 different comment categories. They evaluate their dataset on a multinomial Naive Bayes classifier which serves as our baseline for this task.

Related Work. For a more general survey on self-admitted technical debt see e.g., Sierra *et al.* [46]. Santos *et al.* [5] use a Long short-term memory network (LSTM) to *classify* SATD, also making use of the dataset by S. Maldonado *et al.*; since we confined ourselves to SATD *detection* this work was not included. A text-mining based approach to SATD detection can be found in [47], [48]; in this work, unfortunately, only a subset of the dataset was used and results are thus not comparable. In a closely related comment classification work, Pascarella [49] focus on comment classification in mobile applications.

2.3 Datasets of Source Code

These datasets significantly differ from natural language: on the one hand, code has a very specific and unambiguous syntax and are much more repetitive than natural language [50]; on the other hand, code has very complex semantics, and has many identifiers, leading to vocabulary issues [51].

Code-Comment Coherence Prediction. Corazza *et al.* [24], [52] and Cimasa *et al.* [53] examine the concept of code-comment coherence, i.e., the "relatedness" of a method's code and its lead comment. The authors introduce a dataset of 2,883 *Java* methods along with their leading comment and a binary label indicating whether coherence exists between the two or not [24]. In follow-up work, the authors trained an SVM classifier on their dataset using features based on tf-idf [52] and later word embedings [53]. We include both of these models as baselines.

Linguistic Smell Detection. Fakhoury *et al.* [2] examine the automatic detection of *linguistic* code smells (also known as linguistic antipatterns), that is, code smells emerging from the use of misleading identifier names or the violation of common naming conventions. Examples of this are variable names as if they were lists or arrays when in fact they have a scalar type, or getter methods with side effects.

They labeled a dataset of roughly 1,700 code snippets, following a taxonomy of linguistic smells [28], which comprises 18 different types of linguistic antipatterns. They then trained a number of models on this dataset and compared their performances. These models include CNNs in various configurations, SVMs with different kernels and a Random Forest classifier. All models are binary, that is, they only determine whether a given sample is "smelly" or not and do distinguish between different types of antipatterns. Interestingly, the authors found that a thoroughly tuned SVM model outstrips the CNN in all its configurations. This not only in terms of performance metrics but also in terms of memory consumption, training time and ease of use. Their best-performing models were selected as baselines.

Code Runtime Complexity Classification. Sikka *et al.* [25] investigate the use of machine learning to automatically predict the code runtime complexity class (e.g., $\mathcal{O}(n^2)$) of short programs. To this end, they collected 933 Java implementations of various algorithms from a competitive programming platform and annotated each with the corresponding complexity class (i.e., one of $\mathcal{O}(1)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n)$, $\mathcal{O}(n\log n)$, $\mathcal{O}(n^2)$). They experiment with various traditional machine learning approaches, training, such as Random Forests and SVMs on manually engineered features (such as numbers of loops, numbers of variables etc.) and code embeddings obtained from the programs' abstract syntax tree through graph2vec [54].

Code Readability Prediction. What constitutes readable code and what does not, seems to be largely a matter of personal taste. Notwithstanding this, research by Buse and Weimer [55] suggests that code readability can, at least in part, be measured objectively.

A relatively small number of papers [26], [55], [56], [57], [58] examine models for automatic code readability estimation. Most recently, Scalabrino *et al.* [26], [58] compiled a dataset by letting 30 Computer Science students rate the readability of 200 methods, previously selected from well-known Java projects. Each method received 9 readability ratings: these ratings were then averaged and compared against a threshold value to assign a single binary readability label. Further, they developed a logistic regression model for code readability estimation by combining structural readability features proposed in Buse and Weimer [55] and Dorn [57] with novel textual features. We base all of our experiments on above dataset; previous datasets by Buse and Weimer [55] and Dorn [57] were not included due to lack of baselines suitable for comparison.

2.4 Uses of Pre-Training and Transformers in Software Engineering

In previous work, we investigated the usefulness of the pre-training paradigm (using the earlier ULMFit approach [8]), finding it promising in limited sentiment analysis experiments [13]. Mahadi *et al.* [14] experimented with cross-dataset classification of design discussions, but had mixed results.

Zhang *et al.* [15] provide a detailed study on the use of Transformers for sentiment analysis of Software Engineering artifacts, comparing existing sentiment analysis tools with Transformer models (BERT, RoBERTa, XLNet). Biswas *et al.* [17] pursue a similar avenue, training BERT on a newly compiled dataset of 4,000 sentences from Stack Overflow discussions and comparing results with recurrent models.

Hey *et al.* [16] present NoRBERT, a BERT model fine-tuned to classify functional and non-functional requirements that achieves results competitive to state-of-the-art models on the PROMISE NFR dataset.

Keim *et al.* [59] attempt to use a standard BERT (i.e., pre-trained on English natural language) for the detection of

architectural tactics in Java code and report mixed results that lag behind state-of-the-art approaches in one case study.

Svyatkovskiy *et al.* [60] introduce IntelliCode compose, a system for intelligent code completion based on the GPT-2 Transformer language model.

Finally, Feng *et al.* [61] present CodeBERT, a RoBERTa-based Transformer that was trained on natural language and code (bimodal), allowing for code-related tasks that also involve natural language, such as code search or documentation generation.

To the best of our knowledge, none of the previous work has studied the performance of pre-trained and fine-tuned Transformers on a variety of different small Software Engineering datasets, under different training regimens as well as in addition to techniques such as intermediate fine-tuning, self-training, etc. In particular, for our code experiments we used models that have been pre-trained or fine-tuned on large code corpora, and we also use software-specific pre-training corpora for other tasks (Stack Overflow, app reviews).

3 Background

This section provides an overview of the various machine learning techniques that we investigate in order to evaluate their effectiveness starting with the pre-training paradigm, then covering self-training, data augmentation, active learning, and soft labels. We also highlight the uses of pre-training in software engineering.

3.1 Pre-Training With Transformers, BERT, and RoBERTa

The Transformer Architecture. Transformer [62] networks are a relatively recent architecture, particularly popular in the domain of natural language processing (NLP). They have replaced Long short-term memory (LSTM) networks as the prevailing architecture for text-based data. Transformers are based on attention, a mechanism previously used in LSTM networks to align the information flow between the encoder and decoder part of the network [63]. Attention allows a model to connect related parts of a sentence and form complex structures of interdependence between them. Unlike LSTMs, Transformer networks are not recurrent and instead have a fixed-size input window of tokens (typically 512 tokens): this allows for more efficient training and avoids vanishing-gradient or long-dependency problems extant in recurrent architectures. To summarize, each layer of the Transformer uses the attention mechanism to learn relationships between its inputs, which, in the case of the first layer are the input tokens; when used for classification, a final fully-connected layer is used as output layer.

Pre-Training via Language Modelling. BERT [9] (Bi-Directional Encoder Representations from Transformers) is an extension of the Transformer architecture and comes with a specific *semi-supervised learning* training regimen: BERT heavily relies on *pre-training*, a form of *unsupervised learning*, before being *fine-tuned* on a downstream task in a classical *supervised* fashion.

During pre-training, BERT is trained on large amounts of unlabeled data via Mask Language Modelling (MLM). MLM is a prediction task where some of the input tokens are randomly replaced by blanks ("masked") and the model is trained to predict the tokens behind these blanks, taking into account the textual context on both sides of the blank (see the BERT paper for more details on the pre-training itself [9]). Intuitively, this general task is supposed to initialize the weights to a state in which certain general concepts and relationships useful for a large number of downstream tasks are already present: BERT learns a *Representation* of the tokens. Unlike word embeddings [64], these are contextual representations: they depend both on the token, and its surrounding tokens.

Of note, earlier work also used Language Modelling as a pre-training task (ELMo and ULMFit [7], [8]) with LSTMs, and were used with some varying amount of success in Software Engineering [13], [14]. BERT's pre-training is more efficient for two reasons: BERT's bidirectional architecture uses the context before and after the token, whereas LSTMs use only the context before the token; and BERT uses Byte-Pair Encoding (BPE) [65] to tokenise text in subwords rather than entire words, leading to better modelling of the vocabulary (see previous work by Karampatsis *et al.* for an extended discussion of this aspect for source code [51]).

RoBERTa [10] is a refinement of BERT, in particular relating to its pre-training regimen (e.g., RoBERTa uses a larger pre-training corpus, dynamic masking, and a variation of the pre-training task) and with only minor architectural changes (RoBERTa uses Byte-level BPE tokenization, rather than character-level BPE).

Fine-Tuning. Both BERT and RoBERTa are hardly ever trained from scratch. Instead, starting from a pre-trained model with pre-initialized weights, the model weights are further *fine-tuned* by training on task-specific labeled data (called a downstream task). This involves replacing the last layer of the model (useful for the pre-training task), with a task-specific layer, and resuming training. The model can leverage the pre-trained representations to be able to learn the downstream task effectively, even with a limited amount of data, allowing BERT and RoBERTa to set the state of the art on NLP benchmarks, even on tasks with limited data (the GLUE benchmark [11] includes several task with less than 10,000 examples).

Impact of the Pre-Training Corpora. The standard BERT and RoBERTa models have both been pre-trained on a large English natural language corpus, with several models available in various sizes. There exist pre-trained BERT models for many other natural languages and even programming languages [61]. Intuitively, one would expect a generic pre-training corpus to be a "jack of all trades, master of none", with a more specific pre-training corpus to be more suited for more specific domains (such as software engineering). There is evidence of this for word embeddings in Software Engineering [66], but how much of an impact a domain-specific pre-training corpus has for a BERT or RoBERTa model is still an open question, which we investigate. Of note, the ULMFit approach [8] continues the pre-training task on the task-specific data (without using labels), before the actual fine-tuning, finding that it does improve performance.

3.2 Additional Techniques to Make the Most of Small Datasets

Intermediate-Task Fine-Tuning. Intermediate-task fine-tuning (ITT), also known as two stage fine-tuning, STILTs [67], or

EN	Leppie, that's great news! I look forward to trying IronScheme!
EN →DE	Leppie, das sind großartige Neuigkeiten! Ich freue mich darauf, IronScheme auszuprobieren!
DE →EN	leppie, those are great news! I am looking forward to try out IronScheme!
EN →FR	Leppie, c'est une excellente nouvelle! J'ai hâte d'essayer IronScheme!
FR →EN	leppie, this is great news! I can't wait to try IronScheme!

Fig. 1. Example of back-translation. The original English sentence is first translated to German and French, then translated back into English; resulting variation underlined. Google Translate was used for the translation.

TANDA [68] is a technique whereby the model is fine-tuned twice (with labeled data): first on an *intermediate task*, a task different from but closely related to the target task, and finally on the actual target task (e.g., training for sentiment analysis on movies, before switching to sentiment analysis on books). This is particularly attractive whenever only little data is available for the target task whilst large amounts of data are available for a similar, possibly slightly simpler, but different intermediate task. The idea is that the target task might benefit from "knowledge" that the model acquired during intermediate-task training. Pruksachatkun *et al.* [69] presents a survey on when this method offers good prospects in NLP.

Self-Training. *Self-training* (also known as self-labelling or self-learning) [70], [71], is a very simple semi-supervised learning method. It can be explained as follows. A model is first trained on a (possibly too small) labeled dataset. Next, this model is used to evaluate a number of additional *unlabeled* samples. The model's predictions for these unlabeled samples are then simply used as their gold labels. We now have additional labeled data, albeit noisier ones; after adding it to the original dataset we retrain the model. Predictions can be filtered by confidence to reduce the probability of introducing noise into the training set.

Data Augmentation and Back-Translation. Data augmentation is a well-known technique to increase the amount of labeled data without any human labeling effort, which is especially valuable in cases where training data is in short supply. It works by adding slightly *varied* copies of already existing, labeled samples to the dataset, assuming the variations do not affect the label. The technique was first used in computer vision, where data augmentations are easier to define, such as flipping images horizontally (a dog looking left instead of right), or cropping images randomly (a closeup of the dog's head should still be classified as a dog). For text data, several such methods for augmentation have been proposed in recent years, among others: a) replacing words with synonyms [72], [73], b) replacing, adding or deleting words randomly [73], c) replacing words with the nearest neighbor in an embedding space [74], [75], d) replacing words with predictions from a masked language model such as BERT [76], e) translating into an intermediate language and then back into the source language (back-translation) [77].

Augmentation is typically applied at training time by simply adding the augmented samples to the training set and then proceeding as usual. Alternatively, augmentation can also be carried out at test time by aggregating (e.g., averaging) the prediction for an original test sample with the predictions for its augmented copies, thus obtaining potentially more stable or more accurate predictions.

Fig. 1 shows an example of augmentation through back-translation: a sample in the dataset (here an English sentence) is translated into German and French, then back into English, causing slight variations.

Active Learning. The goal of active learning is to make the process of manual data labeling more efficient. Active learning avoids presenting samples to the rater that the model is likely to classify correctly and thus provide little new information.

Initially, a human rater labels a small number of samples, called the *seed*. There is also a second, larger set of yet unlabeled samples, called the *pool*. A model is first trained on the seed. In a next step, this model is used to select those samples from the pool that the model found most "difficult" to classify. "Difficulty" is measured by means of a *confidence* or *acquisition function* which calculates a confidence score from the model's prediction. In classification, this is usually a distribution over the target classes and acquisition functions are thus applied to class probabilities. Selecting samples by confidence score is called *confidence sampling*. The rater then labels the selected samples, which are then removed from the pool and added to the model's training set. This process is repeated until a satisfying number of samples have been labeled or the model reaches a particular target accuracy. A possible problem with confidence sampling is that selected samples, albeit being difficult for the model, might all be very similar, reducing the efficiency of the process. Confidence sampling is often paired with *diversity sampling*: selected samples are subsequently filtered for diversity, for instance using a clustering algorithm such as *k*-means. A common way to evaluate and compare active learning approaches is a *simulation* with an already labeled dataset. See Settles [78] for an overview of variants and extensions of active learning.

Soft Labels. In classification, usually, every sample is associated with a single target class. For many machine learning algorithms, in particular for neural networks, the target label of a sample is represented as a probability distribution over classes. While optimizations exist for handling the common single-label case, conceptually we can say that the target label is denoted by a distribution vector which assigns probability one to the class it belongs to and probability zero to all other classes. Take, for example, a classification problem where each sample belongs to either class A, B or C. A sample will have the target vector (A: 0, B: 1, C: 0) if it belongs to class B and (A: 0, B: 0, C: 1) if it belongs to class C.

The term *soft label* is used when this distribution vector is fuzzy, i.e., is not comprised of a single one and many zeros. Intuitively, this means that a sample can belong to multiple classes, with a degree expressed by the class probabilities: a target distribution such as (A: 0.4, B: 0.6, C:0) belongs to both, class B and class A. Most datasets do not come with soft-labels. In cases where each sample in a dataset was classified by multiple raters (which is common in order to compute inter-rater agreement), instead of using a majority vote, the rater's votes can be converted into a soft-label. Intuitively, an example in which raters disagree can be seen as

more ambiguous. Providing this information to the model can help it differentiate between "easy" examples and "hard" examples.

4 Methodology

This section covers general aspects of the methodology, that apply to all the experiments. To ease readability, methodological details that refer to a specific technique (e.g., active learning) are described jointly with the results of this technique in Section 5.

For each of the "additional techniques" we select a suitable dataset. We apply domain specific pre-training to all natural language tasks as here language is rather technical, (i.e., software related) and thus differs slightly from the generic English that was used to pre-train BERT and RoBERTa. For code-related tasks, there is no such linguistic discrepancy and we do not further *pre*-train code models. For our back-translation experiment we select the sentiment and app review classification tasks: these datasets are of high quality and their size (3,000-4,000 examples) is within the capacity of our Google Translate based back-translation pipeline. The same datasets were also selected for the active learning experiments for much of the same reasons.

We select the SATD detection task for self-training. For this task, domain-specific unlabeled samples (i.e., Java code comments) are easy to obtain. For the related comment classification task our models already reach a high recall of close to 90% without any aiding technique; hence, it is not a good candidate, as self-training is supposed to boost recall.

For intermediate task training, having a meaningful and "natural" intermediate task to train on is crucial. Such a task was easily found for comment-code coherence. Instead of coherence, in the intermediate task, the model must detect whether a lead comment truly belongs to the method or was assigned randomly. The corresponding training set was obtained by simply shuffling the lead-comments in a set of Java methods.

Soft-labels were only available for the sentiment classification task, where they could be derived from the multi-rater labels which were released together with the dataset. Label-smoothing was applied to the same dataset: this allowed a comparison between the performance of "true" and "smoothed" soft labels.

Finally, the readability task offered itself for task-specific tokens as way to "inject" hand-engineered features into the input. Our models lagged far behind simpler models using manually crafted features. Previous work found line length to be a particularly effective feature: we thus tried to introduce line length information through special tokens.

4.1 Pre-Trained Models

Off-the-Shelf Models. We use several "off-the-shelf" pre-trained model, which were trained on a corpus of generic English text.

- BERT-base, a 12 layer Transformer model (110 million parameters), pre-trained on a 3.3 billion words from books and Wikipedia [9].
- BERT-large, a 24 layer Transformer model (340 million parameters), pre-trained on the same corpus [9].
- DistillRoBERTa, a 6 layer Transformer model (82 million parameters), a compressed version of the larger RoBERTa model.

Domain-Specific Models. We pre-train a range of Transformers on data from different domains and of different sizes, to gain insights on the effectiveness of pre-training on domain-specific data. One model (StackOBERTflow) is entirely trained on domain-specific data, while the others are "off-the-shelf" models pre-trained on English, that are further pre-trained on some domain-specific data.

- BERT-reviews, a 12-layer (base) BERT model trained on 169,097 (8.4 MB) unlabeled app reviews from the AR-Miner dataset. We employed this model in the informative app review detection and app review classification tasks.
- BERT-comments, a 12-layer (base) BERT model trained on 487,693 (48 MB) comments extracted from well-known Java projects and used for comment-related tasks (SATD and classification).
- BERT-SO-1M and BERT-SO-2M, two 12-layer (base) BERT models trained on one (147 MB) and two millions (304 MB) of Stack Overflow comments, respectively, taken from the *Stack Exchange Data Dump*.[1] These models were used in the SATD, sentiment classification, and informative app review detection tasks.
- BERT-SO-1M-large, a 24-layer (large) BERT model trained on one million Stack Overflow comments (as above), used, however, only in the sentiment classification experiments.
- StackOBERTflow, a 6-layer (small) RoBERTa model trained *from scratch* on 26.2 million Stack Overflow comments (3.6 GB). The Stack Overflow corpus was tokenized using byte pair encoding (BPE) subwords and a large vocabulary size of 52,000. We use this model in the informative app review detection, comment classification, app review classification and sentiment classification tasks.

Source Code Models. For the code tasks, we use the following two models:

- CodeBERTa,[2] a small (6-layer) RoBERTa model trained on the polyglot *CodeSearchNet* [79] source code corpus and released by Hugging Face. The model supports multiple programming languages: Go, Java, Javascript, PHP, Python and Ruby.
- CodeBERT [61], a larger 12-layer model trained on the same corpus, but with a bimodal training regimen: the model takes as input pairs of natural language and code and primarily targets code-related tasks that also involve natural language (e.g., code search or summarization). Since none of our code experiments involves natural language, we use an empty string as natural language input, except for the code-comment coherence task, where, after stripping comment markers, comments are treated as natural language.

Scratch Model. To get a rough estimate of the effect of pre-training, we also train, for the sentiment classification task,

1. https://archive.org/details/stackexchange
2. https://huggingface.co/huggingface/CodeBERTa-small-v1

model fully from scratch. This model has the same architecture as StackOBERTflow (a 6-layer, small RoBERTa model), but is randomly initialized and train *solely* on the training set.

4.2 Preprocessing

Pre-trained models do not require extensive pre-processing, such as stemming or removing stop words. In fact, these may be harmful to performance, as the models were pre-trained on data that was *not* pre-processed. In addition, large neural networks have enough parameter capacity to pick up on subtleties such as word order and negation. Thus for most of the datasets, in line with the practice, we did very little preprocessing. For the sentiment analysis and app review datasets, we used raw, unpreprocessed input. For the app review classification tasks we concatenated the review title and body and prepended the review's rating (a number in the range 1-5). We applied heavier preprocessing for tasks with code comment input. Here, similarly to [22], we removed newlines, comment delimiters (such as `//`, `/*`, `*/`), stripped HTML tags and removed all punctuation except periods and question marks as well as repeated whitespace characters. Our goal was to reduce the length of the comments to fit in the Transformer's input window: each punctuation mark is treated as an additional token, taking away a spot in the window.

Preprocessing was also necessary for some source code tasks. In the code-comment coherence task, we simply took the concatenation of the lead comment and the method body as input. Moreover, we reformated all code files in the complexity prediction dataset using Google's Java code formatter[3] such that, for instance, all of them use the same indentation width. For the remaining dataset and tasks no preprocessing was done.

After this, we used each pre-trained model's tokenizer to properly segment the data in the subword units specific to this model, as each model may have a different vocabulary.

4.3 Dataset Partitioning and Evaluation

We tried to replicate the baseline models' training and evaluation methodology as closely as possible. In particular, we use the same split ratio or number of folds (in case k-fold cross validation was used) as was used to train or evaluate the respective baseline with the only exception of the linguistic code smells task. There, we approximated a leave-out-one cross validation with a 15-fold cross validation, as the former was computationally too expensive for our models. Independent of the different evaluation strategies, we repeat all of our experiments at least three times with varying random seeds and average results to reduce noise.

Sentiment Classification. We trained our models on the predefined training set of the Senti4SD dataset [18], 30% of which we use for validation; the corresponding test set was used for testing. All the remaining sentiment analysis datasets were solely used as test sets (as was done in previous work). Whenever a test set lacked a neutral sentiment class, as was the case for the JIRA issues dataset, we treated *neutral* predictions from the model as negative.

Informative App Reviews. A predefined train-test split is also given for the informative app reviews detection task. Here, the test set is actually larger than the training set (2,000 and 1,000 samples, respectively). We used 15% of the training set for validation.

App Review Classification. We used Monte Carlo cross-validation: we split the dataset in 10 random training and validation partitions with a ratio of 70:30. Reported results are averages over 10 runs.

SATD. We use cross-validation, with a $9 \rightarrow 1$ cross-project setting: we train on 9 out of the 10 total projects; the remaining project acts a test set.

Code-Comment Coherence. The model in [52] was trained on 75% of the dataset while the remaining 25% were used for testing. Because this train-test split was chosen randomly, an exact comparison is not possible. To obtain more stable performance metrics, we re-evaluated this baseline model with three random train-splits and averaged the results. We train our own model in the same way, and use 10% of the training data for validation.

Linguistic Code Smells. As pointed out in the initial work, the leave-one-out cross validation strategy used for training the linguistic smell detection baselines is prohibitively expensive for a deep neural network. We resorted to 15-fold cross validation, putting the baselines at a slight advantage: they were trained on over 99% (all but one example) of the entire dataset, while our model uses only 93% (14/15th) of the data for training.

Other Datasets. Finally, the code runtime complexity prediction, comment classification and code readability prediction datasets were evaluated using k-fold cross-validation with k equal to 5, 10 and 10 respectively.

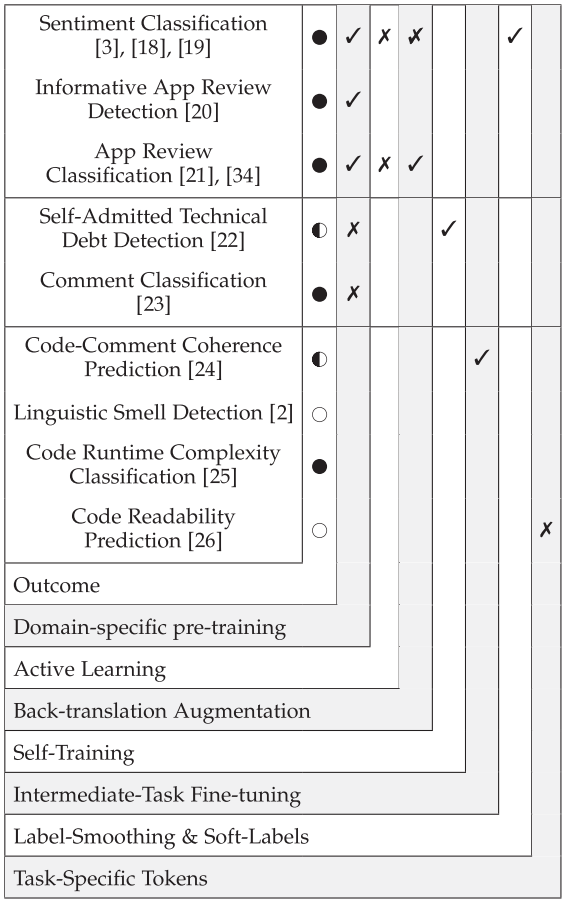
4.4 Fine-Tuning and Testing

We tried several different hyper-parameters on the Senti4SD dataset, varying learning rate (2e-5, 4e-5, 5e-5, 5e-5), batch size (8, 12, 16, 32), and drop-out rate (0.05, 0.07, 0.1). We found that general recommendations give good results in most cases (e.g., a learning rate of 5e-5 for fine-tuning BERT). Interestingly, on Senti4SD a configuration with a relatively small batch size of 12 worked best. On the remaining tasks batch size was selected so as to fill the available GPU memory[4] (16-48 depending on GPU and task). This means that for medium-sized models, the batch size was halved with respect to small models, such as CodeBERTa or StackOBERTflow. We also reduced the Transformers input window size from 512 to 256 tokens on datasets where input sentences where so short that the bulk of them (>98%) fit this narrower window; a smaller window reduces memory consumption (and thus allows for a larger batch size) and speeds up training. Exceptionally long samples that occurred occasionally were truncated to the used window size. For the natural language

3. https://github.com/google/google-java-format

4. the sentiment classification and SATD tasks were carried out on a NVIDIA V100 GPU with 32GB of memory; the remaining experiments on two NVIDIA RTX 2080TI with a total of 20GB of memory.

TABLE 2
Overview of Tasks, Experiments, and Results in This Work



◯ = *not competitive* ◐ = *competitive* ● = *improving* ✗= *no clear benefit* ✗= *little benefit* ✓= *likely benefit. We consider an experiment's outcome as* improving (●) *if our best model's performance is more than 1% above the baseline's for a given task-related metric,* competitive (◐) *if it is within 1% of the baseline's performance and* not competitive (◯) *otherwise. A technique's benefit depends not only on performance improvement but also on effort and applicability and is further discussed in Section 6.*

tasks, truncation affected less than 2% of examples. However, more truncation was needed for code tasks. For the code complexity task, where a single example consists of a whole Java class definition, up to 36% of the examples did not fully fit into the window and had to be truncated. Truncation was also relatively frequent in the code readability and code coherence tasks. In the former, 8% of the examples were truncated (but less than 3% of total tokens); 6% of examples in the latter. We stopped training after validation performance converged, which usually happened after 4-6 epochs.

5 Results and Discussion

We first start by giving an overview of the performance on each task compared to the available baseline, before diving into the details of the impact on the performance of each of the techniques that we investigate. Of note, due to limitations in the datasets and the run-time needed for each experiment, we were limited in the number of experiments we could run for each additional technique. For an overview of our results, refer to Table 2. For reasons of comparability, we use the same metrics[5] that were used for the evaluation of the baselines.

We try to give answers to the following research questions:

RQ1. For which domains and tasks does the pre-training paradigm outperform the baselines? (RQ 1.1) and which pre-training regimen is most effective? (RQ 1.2)

RQ2. Which additional techniques are effective, and if so in which circumstances? In particular we ask RQ2.1: can multi-rater and soft labels be leveraged for better model performance?, RQ2.2: is back translation a suitable remedy for small datasets?, RQ2.3: how effective is active learning for selected Software Engineering tasks, RQ2.4: how can self-training be leveraged to improve model performance? RQ2.5: can intermediate task training raise model performance?

5.1 Comparison With Baselines (RQ1.1)

Sentiment Analysis. On the sentiment analysis datasets, Transformers are ahead of previous methods on most datasets (Table 3). In particular, this is also true for the slightly out-of-domain datasets, such as the JIRA dataset, which the Transformers were not directly trained on, with one exception: Transformers lag behind SentiStrength on the second Stack Overflow test set, but only in terms of F1 (by less than 1%), not accuracy.

App Review Analysis. In both, app review classification and informative app review detection Transformer models clearly outperform baselines (Table 4). The BERT model that was further pre-trained on app reviews is in the lead, but all of the Transformers manage to improve upon baselines. Finally, also on the CLAP dataset StackOBERTflow-comments was able to achieve a 5% higher macro F1 score over the previous random forest model proposed by Scalabrino *et al.* [34] (Table 10).

SATD. The Transformer models are able to outperform the *CNN* model [4]. They remained, however, slightly behind HATD [44], an ELMo-based [45] model which represents the current state of the art for SATD detection (see Table 5). Both HATD and Transformers clearly outperform the other models; both leverage pre-training. The dataset contains a considerable number of exact duplicates and near duplicates (those arising after preprocessing): we report results with and without removal of such duplicates; we do not know whether baseline have been trained with or without such duplicates.

Comment Classification. The Naive Bayes baseline lags behind all three Transformer models (StackOBERTflow-comments, standard BERT, and a domain-specific pre-trained BERT) by a margin of 4% (Table 6). A BERT model further pre-trained on task-specific data (i.e., Java comments) performed slightly worse than standard BERT.

Code-Comment Coherence. The SVM baseline by Corazza *et al.* [52] performs better than the CodeBERTa Transformer, even when employing intermediate-task training (+1%

5. $Acc. = \frac{TP+TN}{TP+TN+FP+FN}$; $Prec. = \frac{TP}{TP+FP}$; $Rec. = \frac{TP}{TP+FN}$; $F1 = \frac{2 \cdot Prec. \cdot Rec.}{Prec.+Rec.}$; for AUC see e.g., [80].

TABLE 3
Accuracy, Macro F1 and Per-Class Precision and Recall for Different Models and Datasets

Dataset	Model	Acc.	F1
Senti4SD [18] (Test Set)	BERT (base)	87.9	87.8
	BERT-SO-1M-large	**89.4**	**89.4**
	BERT-SO-2M	88.8	88.7
	BERT-SO-1M	88.8	88.7
	DistilRoBERTa (small)	87.4	87.3
	RoBERTa (small, no pretr.)	78.4	77.8
	Senti4SD [18]	-	86.0
	SentiCR [18]	-	82.0
	SentiStrengthSE [18]	-	80.0
	SentiStrength [18]	-	84.0
	StackOBERTflow	88.7	88.6
App Reviews [19]	BERT (base)	64.9	50.2
	BERT-SO-1M-large	66.0	51.9
	BERT-SO-2M	67.7	53.1
	BERT-SO-1M	68.3	53.7
	DistilRoBERTa (small)	60.9	48.5
	NLTK [3]	54.0	40.8
	RoBERTa (small, no pretr.)	48.0	42.2
	SentiStrength+AC0-SE [3]	58.9	46.6
	SentiStrength [3]	62.5	48.2
	StackOBERTflow	**72.0**	**57.4**
	Stanford CoreNLP SO [3]	41.6	35.5
	Stanford CoreNLP [3]	69.5	56.0
JIRA Issues [3]	BERT (base)	**95.7**	**95.0**
	BERT-SO-1M-large	94.8	93.8
	BERT-SO-2M	95.2	94.5
	BERT-SO-1M	95.1	94.2
	DistilRoBERTa (small)	93.7	92.7
	NLTK [3]	29.8	46.5
	RoBERTa (small, no pretr.)	84.0	80.1
	SentiStrength+AC0-SE [3]	76.0	87.0
	SentiStrength [3]	77.1	85.4
	StackOBERTflow	93.5	92.5
	Stanford CoreNLP SO [3]	36.0	44.2
	Stanford CoreNLP [3]	67.6	73.7
StackOverflow [19]	BERT (base)	79.9	47.6
	BERT-SO-1M-large	79.2	43.3
	BERT-SO-2M	**80.3**	48.6
	BERT-SO-1M	80.1	47.9
	DistilRoBERTa (small)	78.8	44.3
	NLTK [3]	77.9	43.2
	RoBERTa (small, no pretr.)	75.5	42.2
	SentiStrength+AC0-SE [3]	78.0	46.8
	SentiStrength [3]	69.5	**49.5**
	StackOBERTflow	79.3	44.1
	Stanford CoreNLP SO [3]	75.9	47.5
	Stanford CoreNLP [3]	40.3	35.5

Values reported are means over five runs, each with different seed (only our models). All models were trained on the Senti4SD [18] Stack Overflow dataset.

TABLE 4
Results (Macro F1) for the App Review-Related Datasets

Model	Face-book	Tap Fish	Temple Run2	Swift-Key	Avg.
StackOBERTflow	90.9	89.4	88.6	85.1	88.5
BERT (base)	90.6	88.2	89.2	85.4	88.4
BERT-SO-1M	92.1	90.0	91.1	87.5	90.2
BERT-reviews	**93.3**	**91.4**	**91.3**	**89.9**	**91.5**
EMNB [20]	87.7	76.1	79.7	76.4	80.0

(a) Informative app review detection (AR-Miner)

	Bug reports	Feature request	Ratings	User experience	Avg.
StackOBERTflow	61.1	42.0	79.4	49.8	58.1
BERT (base)	61.8	39.7	79.4	49.4	57.6
BERT-reviews	**64.1**	44.7	**80.2**	51.0	**60.0**
Decision Tree[1] [21]	62.0	42.0	54.0	50.0	52.0
Naive Bayes[1] [21]	62.0	**47.0**	54.0	**53.0**	54.0

[1] multiclass, bag of words + metadata

(b) App review classification

Our numbers are averages over at least three runs with different seeds.

accuracy, Table 7a). In a later work, Cimasa *et al*. [53] experiment with word embeddings: the resulting baseline is weaker than their first and outperformed by the Transformer. As already noted, roughly 6% of examples had to be truncated. When training and evaluating only on the shortest 90% of examples, all entirely fitting the input window, accuracy increased from 81.4 to 83.1%.

Linguistic Smell Detection. We compare with the baselines established by Fakhoury *et al*. [2] in Table 7b. CodeBERTa is able to outperform the manually tuned SVM and the also CNN but clearly remains behind the SMO (sequential minimal optimization) model that was automatically tuned using Bayesian optimization (through Auto-Weka [81]).

Runtime Complexity Classification. The complexity classification task was the only code task where the Transformer exceeded all baselines (Table 7c), including the Random Forest classifier and the SVM trained on AST embeddings.

Dealing with whole Java classes, this task had the largest number of examples that did not fit the Transformer's input window. We found that truncation has a profound effect on performance. We sorted data by length and separately evaluated only on the lower and upper half. 84.6% accuracy was achieved on the lower half (shorter examples), in which no examples had to be truncated. On the upper half (longer examples), 58% of samples had to be truncated; performance dropped to 65.7%.

Code Readability Prediction. The logistic regression baseline trained on manually engineered features by Scalabrino *et al*. [58] is out of reach for the Transformer: the accuracy achieved by the baseline is over 10% higher (Table 7d). From all the selected tasks, the readability prediction task was the hardest for the Transformer.

> RQ1.1: Pre-trained transformers were able to outperform baselines on domains closer to natural language; for source code, results were mixed.

5.2 Pre-Training (RQ1.2)

Our results suggest that, in particular for natural language tasks, the most promising approach seems to be to further pre-train models already pre-trained on general English. When available, in-domain data should be used for pre-training, but even close-to-domain data can yield good improvements. For instance, pre-training BERT on Stack Overflow comments helped to improve accuracy also on the app review dataset (+1.8 F1, see Table 4). Further pre-

TABLE 5
Macro F1 Scores for the SATD Detection Task

	without duplicates				with duplicates						
	BERT-SO-1M	BERT-comments	BERT (base)	Stack-OBERTflow	BERT-SO-1M	BERT-comments	BERT (base)	Stack-OBERTflow	CNN [4]	NLP [22]	HATD [44]
Apache Ant	70.2	66.9	65.4	67.5	70.3	68.8	67.2	69.0	66.0	51.2	**71.3**
ArgoUML	89.8	90.0	89.7	89.3	89.4	90.0	89.7	89.0	87.8	81.9	**90.3**
Columba	90.9	90.4	91.0	90.0	91.4	91.4	91.9	90.6	85.2	75.0	**92.4**
EMF	73.5	72.4	73.2	69.2	72.5	69.5	74.4	71.0	67.9	46.2	**76.5**
Hibernate	88.6	87.5	88.2	87.4	88.7	88.3	88.9	87.7	82.6	76.3	**89.9**
JEdit	72.7	70.6	71.9	73.9	72.0	70.8	70.5	72.2	59.9	46.1	**82.6**
JFreeChart	77.7	**80.7**	79.2	77.6	64.1	64.7	63.8	63.0	73.9	51.3	70.1
JMeter	87.0	**87.5**	87.4	86.6	86.4	85.8	86.2	84.3	82.8	71.5	84.4
JRuby	91.1	91.2	90.8	90.5	92.1	**92.4**	92.3	91.4	86.3	77.3	91.8
SQuirrel	79.1	78.2	78.8	78.6	80.5	79.5	80.5	78.4	73.9	59.3	**80.6**
Average	82.1	81.5	81.6	81.1	80.7	80.1	80.5	79.7	76.6	63.6	**83.0**

As far as our results are concerned, numbers are means over five runs, each with different seed.

training an already pre-trained model should also be preferred over pre-training from scratch. The further pre-trained models outperformed our model pre-trained from scratch for most tasks and metrics even though pre-training from scratch required considerably more training time and (unlabeled) training data. This comparison comes with a grain of salt: our further pre-trained models have twice as many layers as our pre-trained-from-scratch model, which, in turn, has a much larger vocabulary (52,000 versus 30,522). While not having a larger model pre-trained from scratch is a limitation of this work, it also highlights how expensive it is.

What speaks for our small model, and for small models in general, is of course their size: with only half the layers, training and evaluation is roughly twice as fast, the memory footprint is much smaller and, depending on the task, the performance hit may be acceptable.

Our experiments also indicate that further pre-training is effective even with relatively small amounts data. In the sentiment classification task as little as 150 MB (1 million samples) of pre-training data seems to be sufficient and able to "saturate" the model. Doubling the amount of pre-training data resulted in virtually negligible improvements (see BERT-SO-1M versus BERT-SO-2M in Table 3). Similarly, for our large model (BERT-SO-1M-large) improvements are marginal: on the Senti4SD test set, i.e., the test set that "matches" the training set, it outperforms the base-sized models by only 0.6%, while on the other test tests it lags behind them.

While domain-specific pre-training was also effective on the app review datasets where it boosted F1 between +2.4 and +3 (Table 4), this was not the case for code comment tasks. Interestingly, our BERT-comments model, a general English model further pre-trained on Java comments, performs slightly worse than the same model without this task-specific pre-training (i.e., a standard BERT) on both datasets it was applied to (-0.1 F1, see Tables 6 and 5). As to why this is the case we can only *speculate*: A possible explanation is that the comments in our pre-training dataset are very repetitive and have low linguistic diversity (e.g., Java docstrings). Thus, the model might have unlearned some of its general language capabilities during task-specific pre-training.

Fig. 2 demonstrates that pre-training is essential: a randomly initialized model not only converges much slower, it also has higher variance and typically reaches much lower peak performance. In sum, our experiments show that there is very little reason not to use an already pre-trained, general natural language model as the basis for further domain-specific pre-training and should in most cases be

TABLE 6
Results for the Comment Classification Task

Model	F1	Prec.	Rec.
StackOBERTflow	88.4	89.6	**90.1**
BERT-comments	88.3	**90.6**	89.0
BERT	**88.4**	90.5	89.1
Naive Bayes Multinomial [23]	84.3	82.0	87.2

Task-specific pre-training failed to improve performance. As far as our models are concerned, results are averages over three runs with different seeds.

TABLE 7
Results for Code-Related Tasks

Model	Acc.	AUC
CodeBERTa	81.4	80.1
CodeBERTa+ITT	82.5	80.9
CodeBERT	81.6	80.0
SVM [52][1]	**83.5**	81.3
SVM [52][2]	83.3	**82.2**
SVM+WE [53][3]	80.5	-

[1] single seed; value from [52]
[2] average over 5 seeds; reproduction
[3] word embeddings
(a) Comment-code coherence

Model	F1	Prec.	Rec.
CodeBERTa[1]	81.1	81.5	81.1
CodeBERT[1]	71.2	73.7	71.5
SMO Poly[2] [2]	**88.8**	**91.8**	**86.0**
SVM RBF[2] [2]	74.8	76.2	73.4
CNN[2] [2]	74.5	75.6	73.5

[1] 15-fold cross validation
[2] leave-one-out cross validation
(b) Linguistic smell detection

Model	Acc.
CodeBERTa	78.2
CodeBERT	**78.4**
Random Forest [26]	74.3
Logistic Regr. [26]	73.2
SVM [26]	73.0
SVM+graph2vec [26]	73.9

(c) Code runtime complexity prediction

Model	Acc.
CodeBERTa	73.1
CodeBERTa+LLT	72.5
CodeBERT	69.3
Logistic Regr. [58]	**84.0**

(d) Readability prediction

As far as our number are concerned, values are means over at least three runs with different seeds.

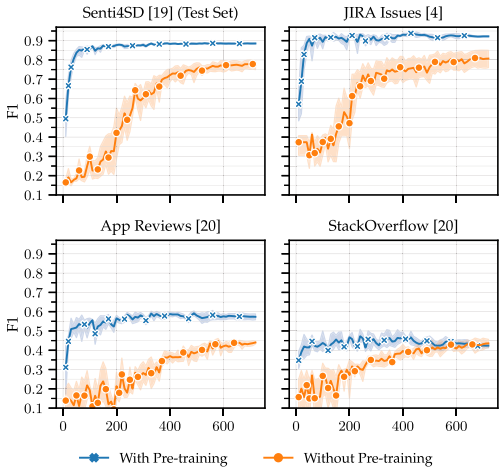


Fig. 2. F1 score on different sentiment classification datasets with and without pre-training. Number of optimization steps is shown on the x-axis; error bands are 95% confidence intervals.

preferred over pre-training from scratch, which, in relation to training time, hardly seems worth the effort.

RQ1.2: Pre-training is essential. Further pre-training a generic model on domain-specific data is often beneficial, and is much more effective than pre-training from scratch.

5.3 Soft Labels (RQ2.1)

Since Calefato *et al*. [18] released multi-rater labels (three per sample) along with the majority label, we conducted a soft label experiment. For instance, if one voter assigned the *positive* label to a sentence, while two raters assigned *neutral*, majority voting would label it as *neutral*. Instead, the soft label captures all three rater labels, assigning the distribution: (positive: $0.\overline{33}$, negative: 0, neutral: $0.\overline{66}$) to the sentence.

Method. We train a subset of the models with hard-labels and soft-labels on Senti4SD, and evaluate on all sentiment analysis datasets (see Table 8).

Result. On the Senti4SD test set, training with all three rater labels resulted in an increase of 0.5%. On the JIRA test set, soft labels yielded an improvement of 1%. On the other hand, performance dropped on the second Stack Overflow test set. While these improvements are not certain, and might seem modest, they come almost for free. Whenever multi-rater labels are available, we recommend to tentatively use them in this way and encourage creators of datasets to also release labels of individual raters.

RQ2.1: When available, individual rater labels may improve performance at very low cost. We hope this result will encourage more researchers to release them.

5.4 Back-Translation Augmentation (RQ2.2)

We performed back-translation experiments on the sentiment and app review classification tasks by translating the entire datasets into French, German and Russian using Google Translate and from these languages back into English (see Fig. 1 for an example).

Method. For the CLAP and Senti4SD datasets we do training-time and test-time augmentation, both, separately and combined, using the StackOBERTflow model (Table 10). On the other app review dataset [21] we do training-time augmentation alone, and combine it with test-time augmentation: here the the experiment is carried out on several different models (Table 9).

Result. Back-translation augmentation led to a clear increase in F1 and accuracy on the CLAP app review dataset, in particular when training and testing time augmentation were combined (+1.1% accuracy, Table 10). On Senti4SD, data augmentation yields modest improvements (+0.3%); in fact, augmenting at test time only caused a slight drop in performance. Table 9 suggests that the effect of back-translation augmentation depends on the model and pre-training choice. With train-time augmentation only, we see a modest increase of 0.6% in F1 for our small StackOBERTflow model and 1.1% on a general BERT model, while BERT-reviews shows better performance without augmentation. The latter does however benefit from combined augmentation (+0.4%). However, the question of whether in general task-specific pre-training diminishes the effects of data augmentation cannot be answered given this limited data and would require further experiments.

TABLE 8
Macro F1 for Different Label Types and Datasets: Mean, Maximum and Standard Deviation Over Five Runs, Each With Different Seed

Dataset	Label Type	F1	
		$\mu \pm \sigma$	max
App Reviews [19]	All Label Votes	**57.5** $\pm$ 2.2	59.1
	Majority Label	57.4 $\pm$ 1.6	59.7
JIRA Issues [3]	All Label Votes	93.5 $\pm$ 1.0	**94.7**
	Majority Label	92.5 $\pm$ 0.5	93.2
StackOverflow [19]	All Label Votes	42.4 $\pm$ 1.0	43.8
	Majority Label	**44.1** $\pm$ 2.9	46.4
Senti4SD [18]	All Label Votes	**89.1** $\pm$ 0.5	89.6
	Majority Label	88.6 $\pm$ 0.5	89.1

All models were trained on the Senti4SD [18] Stack Overflow dataset.

TABLE 9
F1 Scores for App Review Classification With Training-Time Back-Translation Augmentation (+BT) and Training-Time *and* Test-Time Back-Translation Augmentation (+BTT)

	Bug reports	Feature request	Ratings	User experience	Avg.	Change
StackOBERTflow +BT	62.5	43.3	79.0	50.2	58.7	+0.6
StackOBERTflow +BTT	62.7	44.3	79.6	50.6	59.3	+1.2
BERT (base)+BT	62.0	43.1	79.4	50.2	58.7	+1.1
BERT (base) +BTT	62.5	42.6	80.0	50.1	58.8	**+1.2**
BERT-reviews +BT	63.2	43.9	79.5	51.0	59.4	-0.6
BERT-reviews +BTT	**63.7**	**46.7**	**80.1**	**51.2**	**60.4**	+0.4

Change is relative to the same model without any augmentation. Results are averages over three runs with different seeds.

TABLE 10
Macro F1, Precision and Recall for Back-Translation Augmentation at Training and Test Time on the Senti4SD (Sentiment Classification) and CLAP (App Review Classification) Datasets Using the StackOBERTflow Model

Dataset	Augmentation	Acc.	F1	Prec.	Rec.
CLAP	Train+Test	**89.2**	**81.4**	**84.2**	**82.3**
	None	88.1	79.7	80.6	81.7
	Test	88.3	80.2	81.8	81.5
	Train	88.9	81.1	83.7	82.0
Senti4SD	Train+Test	**88.7**	**88.6**	**88.7**	**88.5**
	None	88.4	88.2	88.2	88.4
	Test	88.2	88.0	88.1	88.0
	Train	88.4	88.3	88.3	88.5

Results are averages over three runs with different seeds.

RQ2.2: When possible, back translation yields improvements, particularly if used at both training and test time.

5.5 Active Learning (RQ2.3)

We try active learning on the Senti4SD sentiment analysis dataset and an app review dataset. In both cases we compare several acquisition functions (Table 11).

Method. We carry out the experiment as follows: initially we split the *training* set into the seed set $\mathcal{D}_{seed}$ containing 5% of all samples and the pool set $\mathcal{D}_{pool}$, containing the remaining samples. We let $\mathcal{D}_{train} := \mathcal{D}_{seed}$ and train the model. Then we evaluate $\mathcal{D}_{pool}$ as well as the test set on this model. Next, for each $x \in \mathcal{D}_{pool}$ we calculate a confidence score by applying the acquisition function (from Table 11). After that, we let $\mathcal{D}_{top}$ be the $k = 180$ samples with the highest confidence score. We remove these samples from $\mathcal{D}_{pool}$ and add them to $\mathcal{D}_{train}$ This procedure is repeated until $\mathcal{D}_{pool}$ is empty.

We also combine confidence sampling with diversity sampling to avoid introducing similar samples into the pool. Instead of k samples, we select $c \cdot k$ samples from the pool, where c determines the cluster size. We use the k-means algorithm to cluster the $c \cdot k$ samples into k clusters, each of size c. We then select a single sample from each cluster, for a total of k samples. Then, we proceed as above.

Result. Fig. 3 shows evaluation results at each iteration step with a cluster size of $c = 3$ for the sentiment classification task. The outcome of our active learning experiments remained behind expectations: in both tasks, neither confidence sampling alone nor confidence and diversity sampling combined showed an appreciable advantage over the random baseline. The choice of acquisition function did not seem crucial, but a more systematic study would be needed to draw more solid conclusions. On the other hand, the plot of pool accuracy (top left) indicates that the active learning process worked as expected: a random acquisition function without diversity sampling had constant performance.

TABLE 11
Acquisition Functions Used in Our Active Learning Experiment, Adapted From [82]: Least Confidence (C_{LC}), Margin of Confidence (C_{MC}), Ratio of Confidence (C_{RC}), Entropy (C_E) and Random Confidence (C_{rand})

$$C_{LC}(x) = \frac{n}{n-1}(1 - P_\theta(y_1^*|x))$$
$$C_{MC}(x) = 1 - (P_\theta(y_1^*|x) - P_\theta(y_2^*|x))$$
$$C_{RC}(x) = \frac{P_\theta(y_1^*|x)}{P_\theta(y_2^*|x)}$$
$$C_E(x) = -\frac{1}{log_2(n)} \sum_{i=1}^{n} P_\theta(y_i|x) log_2(P_\theta(y_i|x))$$
$$C_{rand}(x) = rand([0, 1])$$

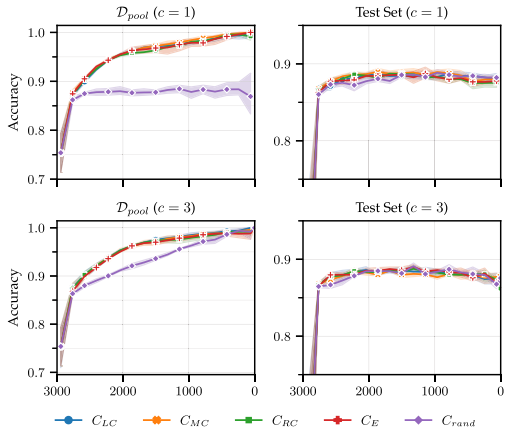


Fig. 3. Accuracy of $\mathcal{D}_{pool}$ and the test set at each iteration of the active learning process for different acquisition functions with ($c = 3$) and without ($c = 1$) diversity sampling for the sentiment classification task (Senti4SD). Error bands are 95% confidence intervals. Same plot for the review classification task can be found in the appendix (Fig. 5), available in the online supplemental material.

RQ2.3: Active learning simulations did not prove successful. Selecting new examples randomly performed as well as selecting examples with low model confidence.

5.6 Self-Training (RQ2.4)

We investigate the use of self-training for the SATD detection task.

Method. We extract 350,000 comments from various popular Java libraries and frameworks. Then we train a classifier model on the entire *original* dataset, which we use to classify the 350,000 comments as either *technical debt* or *not technical debt*. We only keep the 7,904 positive comments, i.e., those classified as *technical debt*, and discard all other samples to avoid increasing the class imbalance already extant in the original dataset. For each positive sample we calculate a confidence score using C_{LC} (Table 11). We take the top 5%, and 80% most confidently classified comments, equal to 6,092 and 7,880 additional comments, respectively, and add them to the original training set. Finally, the model is trained and evaluated on this extended training set.

Result. Fig. 4 shows that self-training increases recall (and possibly F1) but causes precision to drop. This is, of course, not surprising: the added samples increase dataset variance which likely explains an increased recall. Similarly, the precision drop can be explained by the lower quality self-training labels. Thus, one can tune the precision-recall trade-off according to task-specific needs, such as when a recall is more important than precision, or the model's precision is high enough to be partly sacrificed

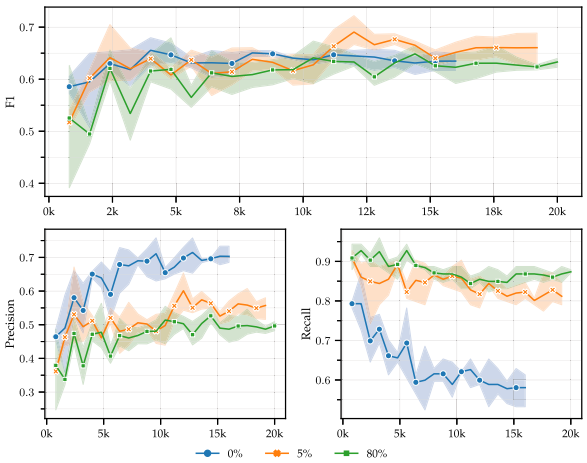


Fig. 4. Precision, recall and F1 under different self-training settings for the *Apache Ant* project. Error bands are 95% confidence intervals.

for better recall. Fig. 4 shows different self-training settings for *Apache Ant*: F1 score went up 4% (precision: −1%, recall +10%), when using a 5% confidence threshold. The change in F1 strongly depends on the confidence threshold and varies across projects: *EMF* sees an 8% F1 drop (precision: −28%, recall: +15%); other projects range from −1% to +4%.

RQ2.4: Self-training increases recall at the expense of precision, but the confidence threshold should be tuned carefully.

5.7 Intermediate-Task Training (RQ2.5)

We evaluate Intermediate-Task Training (ITT) on the code-comment coherence task, as it is the only setting for which we could define such an intermediate task.

Method. We use 38,000 Java methods along with their lead comments from the *CodeSearchNet* [79] dataset. We assign half of the methods to their actual lead comments (assumed to be coherent) and shuffle the other half randomly (thus assumed to be incoherent). The model is then fine-tuned on the intermediate task of detecting whether a method was paired with its true lead comment or a random one. Finally, we fine-tune on the code-comment coherence dataset as usual.

Result. ITT improved the performance of the Transformer model on the code-comment coherence task (Table 7a): we observed a modest rise in AUC (area under the precision-recall curve [80]) of 0.8%, and a slightly higher increase in accuracy (+1.1%). If an appropriate intermediate task for the task at hand can be found, ITT can be done with relatively little effort: in our case, the training procedure for the intermediate-task was mostly identical to the one for the target task; the bulk of the work consisted in generating the intermediate-task dataset (e.g., selecting, shuffling and pre-processing the data from *CodeSearchNet*).

RQ2.5: When applicable, ITT may improve performance; however finding a suitable intermediate task may be difficult.

6 Discussion

6.1 Summary and Implications of the Results

Types of Datasets. Overall, we see that Transformers work very well for natural language datasets, but that performance on source code is "hit or miss". This comes with caveats: the models used so far are multilingual, which might reduce performance. They are also trained with less computational resources, and on an order of magnitude less data: While CodeSearchNet is around 1.7 GB, the training data for BERT is around 16GB, while it is 160GB for RoBERTa (CodeSearchNet). Models where also small (6 layers), or had a dual input (text and code). We would expect a Java-specific model trained on a similar size corpus as BERT to perform better. Moreover, source code is quite different from natural language: code snippets are often larger than sentences, and much more structured. This might pose limits on what an unstructured model with a small window size might achieve. Indeed, many more code samples than natural language samples were truncated to fit the Transformer's window, which we suspect affected performance. Recent adaptations of the Transformer architecture allow the model to better make use of the tree-like structure of code (e.g., [83], [84]), while others allow it to efficiently use a larger window (e.g., [85], [86]). Investigating these architectures in connection with small datasets remains an issue for future research.

Domain-Specific Pre-Training. Proved effective in natural language settings, improving performance at a moderate cost in terms of computation and data. The only case where it did not work well was for code comments. While we are not sure why, one reason could be that code comments are too far way from regular English (needing a specific model instead), or that careful curation of the data set (avoiding too many duplicates) is needed. Both cases could lead to catastrophic forgetting [87] of the initial pre-training. Leveraging the resources that were used to train BERT and fine-tuning it further proved much more effective than training a model from scratch. We have not evaluated domain-specific training from English to source code, as we hypothesized that the two domains are very different—the tokenization alone might differ significantly [51]. This intuition is supported by the literature, which reports an example where an English BERT was applied to source code, with underwhelming results [59].

Data Augmentation. While data augmentation is effective for natural language, it is not immediately applicable to source code. Source code can not be "back translated" easily. Data augmentation for code is largely still an open issue. The only work we are aware of is Jain *et al.* [88] who used various code transformations on JavaScript code for better representation learning. These transformations include e.g., reformatting, dead-code elimination and insertion, as well as variable renaming or transforming for-loops into while-loops.

Intermediate Task Training. Another alternative is to define suitable intermediate training tasks. We have found initial evidence of this, and a recent paper adds further evidence, in the context of traceability [89]. However, it used a very similar task and dataset. Thus, the challenge here is not

whether intermediate task training helps, but rather *whether a suitable task exists* for a given problem.

Soft Labels. Soft labels that reflect the uncertainty of raters (and thus the difficulty of the samples) can be useful as well, and at a minimal cost. However these are not common, as of now. We call on dataset builders to release them alongside the majority label, as was done by Calefato [18].

6.2 Limitations and Threats to Validity

External Validity

Limited Number of Experiments. While we try to report results as extensively as possible to increase their generalisability, we are limited for two main reasons: 1) we have limited computational resources, and 2) some techniques are specific to some settings.

Selected Datasets. We selected 13 datasets from different Software Engineering domains, involving different tasks and types of language (natural language, code comments and source code). While we tried our best to have a diverse set of datasets, we cannot fully preclude that the findings of these work carry over to datasets not used in our experiments, even if similar in nature.

Internal Validity

Limited Resources. Deep learning is famously resource intensive. While fine-tuning is less resource intensive than training models from scratch, it still requires significant time on one or more dedicated GPUs, particularly for larger models. A single run is measured in hours. This limits the number of experiments, particularly as we repeat experiments several times with different random seeds.

Hyper-Parameters. Limited resources also impact the extent to which we perform hyper-parameter optimisation, as thorough parameter searches (whether by grid, random or Bayesian methods) would be prohibitively expensive. A second limit is that some hyper-parameters are fixed by the usage of a pre-trained model (e.g., number of layers, number of attention heads, embedding size, vocabulary size). A silver lining is that, given the interest in pre-trained models, general recommendations for hyper-parameters exist and are broadly applicable. Thus, we started with these recommendations, and investigated some variations of the hyper-parameters on the Senti4SD dataset, confirming that the recommendations worked well. We then applied those hyper-parameters on other experiments, varying only the most important ones in some cases (learning rate, batch size). Cross-validation also makes evaluation and hyper-parameter tuning more complex and resource intensive. Since we limited hyper-parameter tuning, we are not at risk of overfitting to the test fold when doing cross validation. An alternative would be to use doubly nested cross validation, but this further increases the resource needed. We note that dedicated test sets ease this considerably.

Random Seeds. Dodge *et al.* [90] found that the choice of the random seed can have a substantial impact on performance, especially for small datasets. We ran most of our experiments five times and all of them at least three times, with different seeds. While this surely mitigates the problem, it might not fully clear it up.

Comparisons With Previous Work. We do our best to provide a fair comparison with previous work, while avoiding methodological issue (e.g., averaging seeds). We do not always exactly know how previous work was evaluated evaluation (e.g., hyper-parameter selection strategy, whether simple or nested cross-validation was used, or whether some data points were excluded) as code is not always released. In some other cases, other factor presents us to make an exact comparison (e.g., use of leave-one-out cross validation is not practical for our setting). To alleviate this in the future, we release our source code (see Appendix C, available in the online supplemental material).

Active Learning. While we could not see an advantage to active learning, this is not in line with previous work by Dhinakaran *et al.* [42] and Tu *et al.* [91]. Of note, our results are obtained through simulation based on an existing labelled dataset. While this is a practice often used to evaluate active learning methods, a realistic application of active learning on a larger set of unlabelled data would lead to a different training set, which may be substantially more varied, and thus more effective. However the difference in labelled data would have made any comparison with existing work or other presented techniques unfair and thus seemed to not match the theme of this paper. A future study investigating the interaction between active learning and pre-training would be very interesting. It is also possible that the impact of active learning is less visible when pre-training is used.

Implementation Bugs. Our implementations are based on Hugging Face's `transformers` Python package [92], a high quality implementation of common Transformer models. However, despite careful reviews we cannot fully preclude errors in our own code and adaptations.

Transformer Window Size. Our models have a window size of 512 tokens (which is the default for BERT and RoBERTa). While this posed no problem for the studied natural language datasets, where only a few outliers ($<2\%$) exceeded the window size, it did so for the code tasks (6-36%). In particular, when examples are whole classes and not just single methods, as in the runtime complexity task, where up to 36% of examples had to be truncated. Restricting training and evaluation to shorter, non-truncated examples increased performance (+1.7-6%), suggesting that the limited window size is a hindrance for code tasks. However, an alternative explanation for the performance increase could be that shorter examples are simply easier to learn. Recent architectures such as the Longformer [85] or Reformer [86] might remedy this issue in the future.

Construct Validity

Results in Specific Settings. While resources are limited, we still wanted to try each technique on at least two datasets. However, some techniques were applied to a single dataset. For soft labels, we needed multiple ratings: only a single sentiment analysis dataset had the required three ratings per sample. We could only define a reasonable intermediate task for code-comment readability prediction. We considered using self-training for comment classification, but did not, due to the large number of imbalanced classes.

7 Conclusion

Software Engineering datasets are often small, by necessity. In this work, we trained various Transformer models on 13

small and medium-sized dataset selected from the recent Software Engineering literature. We not only compared Transformers of different size and different pre-training regimes but also applied several machine learning techniques that promised a possible benefit for small datasets. These techniques were data augmentation, self-training, intermediate-task training, active learning and soft labels.

Overall, we found that on natural language tasks, Transformers usually outperform existing baselines. On source code tasks, however, results were mixed. Significant work lies ahead to define effective pre-trained source code models either by training larger models on more data, or by incorporating more structural information during training. For some source code tasks, training models with larger input windows than 512 tokens might be beneficial; however these models are not widely available for source code at this time of writing.

In general, we advise *against* pre-training a new model from scratch as it is extremely resource intensive, for mixed results. Instead, an already pre-trained model can be *further pre-trained* on task-specific data. If such task-specific data is unavailable, training on close-to-domain data is worth a try. We provide several such pre-trained models in Appendix C, available in the online supplemental material.

Several additional techniques were useful at a relatively low cost. We particularly recommend the use of soft labels derived from multi-rater labels if available, and call on dataset authors to release these multi-rater labels. Back-translation is similarly useful, if more expensive. It is unfortunately not easily applicable to source code.

Other techniques were less applicable. We find that self-training is advisable only in cases where the user wants to boost recall and is willing to sacrifice precision. If circumstances allow it, intermediate task training seems promising, but it seems rarely applicable, and has a much higher cost. Finally, our active learning experiments were inconclusive; a wider study on a larger set of dataset might be required to draw a clearer picture.

While these general guidelines are useful on their own, their applicability is limited. To this extent, we release all the scripts and pretrained models that were built as part of this work, so that the community can easily fine-tune the models on their own Software Engineering datasets, and apply additional techniques as they see fit (see Appendix C, available in the online supplemental material).

Acknowledgments

We thank the original authors of the datasets that we use; this work would not have been possible without them. We also thank the anonymous reviewers for their comments that greatly improved this paper.

References

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[2] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?," in *Proc. IEEE 25th Int. Conf. Softw. Anal. Evol. Reeng.*, 2018, pp. 602–611.

[3] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: How far can we go?," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 94–104.

[4] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, pp. 1–45, 2019.

[5] R. M. Santos, M. C. R. Junior, and M. G. de Mendonça Neto, "Self-admitted technical debt classification using LSTM neural network," in *Proc. 17th Int. Conf. Inf. Technol.–New Gener.*, 2020, pp. 679–685.

[6] X. Zhu and A. B. Goldberg, "Introduction to semi-supervised learning," *Synthesis Lectures Artif. Intell. Mach. Learn.*, vol. 3, no. 1, pp. 1–130, 2009.

[7] M. E. Peters *et al.*, "Deep contextualized word representations," 2018, *arXiv:1802.05365*.

[8] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," 2018, *arXiv:1801.06146*.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technologies*, 2019, pp. 4171–4186.

[10] Y. Liu *et al.*, "RoBERTa: A robustly optimized BERT pretraining approach," 2019.

[11] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," 2019.

[12] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," 2016.

[13] R. Robbes and A. Janes, "Leveraging small software engineering data sets with pre-trained neural networks," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., New Ideas Emerg. Results*, 2019, pp. 29–32.

[14] A. Mahadi, K. Tongay, and N. A. Ernst, "Cross-dataset design discussion mining," in *Proc. IEEE 27th Int. Conf. Softw. Anal. Evol. Reeng.*, 2020, pp. 149–160.

[15] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, "Sentiment analysis for software engineering: How far can pre-trained transformer models go?," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, pp. 70–80.

[16] T. Hey, J. Keim, A. Koziolek, and W. F. Tichy, "NoRBERT: Transfer learning for requirements classification," in *Proc. IEEE 28th Int. Requirements Eng. Conf.*, 2020, pp. 169–179.

[17] E. Biswas, M. E. Karabulut, L. Pollock, and K. Vijay-Shanker, "Achieving reliable sentiment analysis in the software engineering domain using BERT," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 162–173.

[18] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1352–1382, 2018.

[19] B. Lin, F. Zampetti, R. Oliveto, M. Di Penta, M. Lanza, and G. Bavota, "Two datasets for sentiment analysis in software engineering," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 712–712.

[20] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "AR-Miner: Mining informative reviews for developers from mobile app marketplace," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 767–778.

[21] W. Maalej, Z. Kurtanović, H. Nabil, and C. Stanik, "On the automatic classification of app reviews," *Requirements Eng.*, vol. 21, no. 3, pp. 311–331, 2016.

[22] E. d. S. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1044–1062, Nov. 2017.

[23] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 227–237.

[24] A. Corazza, V. Maggio, B. Kessler, and G. Scanniello, "A new dataset for source code comment coherence," in *Proc. 3rd Italian Conf. Comput. Linguistics*, 2016, Art. no. 100.

[25] J. Sikka, K. Satya, Y. Kumar, S. Uppal, R. R. Shah, and R. Zimmermann, "Learning based methods for code runtime complexity prediction," in *Proc. Eur. Conf. Inf. Retrieval*, 2020, pp. 313–325.

[26] S. Scalabrino, M. Linares-V ásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *Proc. IEEE 24th Int. Conf. Program Comprehension*, 2016, pp. 1–10.

[27] D. Wang, Y. Guo, W. Dong, Z. Wang, H. Liu, and S. Li, "Deep code-comment understanding and assessment," *IEEE Access*, vol. 7, pp. 174 200–174 209, 2019.

[28] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empir. Softw. Eng.*, vol. 21, no. 1, pp. 104–158, 2016.

[29] N. Borovits *et al.*, "DeepIaC: Deep learning-based linguistic anti-pattern detection in IaC," in *Proc. 4th ACM SIGSOFT Int. Workshop Mach.-Learn. Techn. Softw.-Qual. Eval.*, 2020, pp. 7–12. [Online]. Available: https://doi.org/10.1145/3416505.3423564

[30] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, 2016.

[31] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?," in *Proc. IEEE 25th Int. Conf. Softw. Anal. Evol. Reeng.*, 2018, pp. 612–621.

[32] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "On the feasibility of transfer-learning code smells using deep learning," 2019, *arXiv:1904.03031*.

[33] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowl.-Based Syst.*, vol. 128, pp. 43–58, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705117301880

[34] S. Scalabrino, G. Bavota, B. Russo, M. D. Penta, and R. Oliveto, "Listening to the crowd for the release planning of mobile apps," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 68–86, Jan. 2019.

[35] M. Ortu *et al.*, "The emotional side of software developers in JIRA," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 480–483.

[36] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 14–24.

[37] M. V. Mantyla, D. Graziotin, and M. Kuutila, "The evolution of sentiment analysis—A review of research topics, venues, and top cited papers," *Comput. Sci. Rev.*, vol. 27, pp. 16–32, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574013717300606

[38] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized autoregressive pretraining for language understanding," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 5753–5763. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf

[39] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite BERT for self-supervised learning of language representations," 2020.

[40] T. Ahmed, A. Bosu, A. Iqbal, and S. Rahimi, "SentiCR: A customized sentiment analysis tool for code review interactions," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 106–111.

[41] Z. Chen *et al.*, "Emoji-powered sentiment and emotion detection from software developers' communication data," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Jan. 2021, Art. no. 18.

[42] V. T. Dhinakaran, R. Pulle, N. Ajmeri, and P. K. Murukannaiah, "App review analysis via active learning: Reducing supervision effort without compromising classification accuracy," in *Proc. IEEE 26th Int. Requirements Eng. Conf.*, 2018, pp. 170–181.

[43] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spinola, "Towards an ontology of terms on technical debt," in *Proc. 6th Int. Workshop Manag. Tech. Debt*, 2014, pp. 1–7.

[44] X. Wang, J. Liu, L. Li, X. Chen, X. Liu, and H. Wu, "Detecting and explaining self-admitted technical debts with attention-based neural networks," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 871–882.

[45] M. E. Peters *et al.*, "Deep contextualized word representations," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technologies*, 2018, pp. 2227–2237.

[46] G. Sierra, E. Shihab, and Y. Kamei, "A survey of self-admitted technical debt," *J. Syst. Softw.*, vol. 152, pp. 70–82, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219300457

[47] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empir. Softw. Eng.*, vol. 23, no. 1, pp. 418–451, 2018.

[48] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "SATD detector: A text-mining-based self-admitted technical debt detection tool," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.: Companion*, 2018, pp. 9–12.

[49] L. Pascarella, "Classifying code comments in Java mobile applications," in *Proc. IEEE/ACM 5th Int. Conf. Mobile Softw. Eng. Syst.*, 2018, pp. 39–40.

[50] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[51] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= Big vocabulary: Open-vocabulary models for source code," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1073–1085.

[52] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: A dataset and an empirical investigation," *Softw. Qual. J.*, vol. 26, no. 2, pp. 751–777, Jun. 2018.

[53] A. Cimasa, A. Corazza, C. Coviello, and G. Scanniello, "Word embeddings for comment coherence," in *Proc. 45th Euromicro Conf. Softw. Eng. Adv. Appl.*, 2019, pp. 244–251.

[54] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," 2017.

[55] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 546–558, Jul./Aug. 2010.

[56] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 73–82.

[57] J. Dorn, "A general software readability model," 2012.

[58] S. Scalabrino, M. Linares-V ásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *J. Softw.: Evol. Process*, vol. 30, no. 6, 2018, Art. no. e1958.

[59] J. Keim, A. Kaplan, A. Koziolek, and M. Mirakhorli, "Does BERT understand code? – An exploratory study on the detection of architectural tactics in code," in *Proc. Eur. Conf. Softw. Archit.*, 2020, pp. 220–228.

[60] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1433–1443.

[61] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," 2020.

[62] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[63] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv*.

[64] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013, *arXiv:1310.4546*.

[65] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 1715–1725.

[66] V. Efstathiou, C. Chatzilenas, and D. Spinellis, "Word embeddings for the software engineering domain," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 38–41.

[67] J. Phang, T. Févry, and S. R. Bowman, "Sentence encoders on stilts: Supplementary training on intermediate labeled-data tasks," 2018, *arXiv:1811.01088*.

[68] S. Garg, T. Vu, and A. Moschitti, "TANDA: Transfer and adapt pre-trained transformer models for answer sentence selection," in *Proc. AAAI Conf. Artif. Intell.*, 2020, pp. 7780–7788.

[69] Y. Pruksachatkun *et al.*, "Intermediate-task transfer learning with pretrained models for natural language understanding: When and why does it work?," 2020.

[70] H. Scudder, "Probability of error of some adaptive pattern-recognition machines," *IEEE Trans. Inf. Theory*, vol. 11, no. 3, pp. 363–371, Jul. 1965.

[71] D. Yarowsky, "Unsupervised word sense disambiguation rivaling supervised methods," in *Proc. 33rd Annu. Meeting Assoc. Comput. Linguistics*, 1995, pp. 189–196. [Online]. Available: https://www.aclweb.org/anthology/P95–1026

[72] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," 2016.

[73] J. Wei and K. Zou, "EDA: Easy data augmentation techniques for boosting performance on text classification tasks," 2019.

[74] X. Jiao *et al.*, "TinyBERT: Distilling BERT for natural language understanding," in *Proc. Findings Assoc. Comput. Linguistics*, 2020, pp. 4163–4174.
[75] W. Y. Wang and D. Yang, "That's so annoying!!!: A lexical and frame-semantic embedding based data augmentation approach to automatic categorization of annoying behaviors using #petpeeve tweets," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2015, pp. 2557–2563. [Online]. Available: https://www.aclweb.org/anthology/D15–1306
[76] S. Garg and G. Ramakrishnan, "BAE: BERT-based adversarial examples for text classification," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2020, pp. 6174–6181.
[77] R. Sennrich, B. Haddow, and A. Birch, "Improving neural machine translation models with monolingual data," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016.
[78] B. Settles, "Active learning literature survey," Comput. Sci. Dept. Univ. Wisconsin–Madison, Madison, WI, USA, Tech. Rep. 1648, 2009.
[79] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv:1909.09436*, Sep. 2019. [Online]. Available: http://arxiv.org/abs/1909.09436
[80] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to Information Retrieval*, vol. 39. Cambridge, U.K.: Cambridge Univ. Press Cambridge, 2008.
[81] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-WEKA: Automatic model selection and hyperparameter optimization in WEKA," in *Automated Machine Learning*. Cham, Switzerland: Springer, 2019, pp. 81–95.
[82] R. Munro, *Human-in-the-Loop Machine Learning*. Shelter Island, NY, USA: Manning Publications Co., 2021.
[83] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," 2020, *arXiv:2003.13848*.
[84] V. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 12 081–12 091.
[85] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 2020.
[86] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," 2020.
[87] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," 2015.
[88] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," 2020, *arXiv*.
[89] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability transformed: Generating more accurate links with pre-trained BERT models," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 324–335.
[90] J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. Smith, "Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping," 2020, *arXiv:2002.06305*.
[91] H. Tu, Z. Yu, and T. Menzies, "Better data labelling with EMBLEM (and how that impacts defect prediction)," *IEEE Trans. Softw. Eng.*, to be published, Apr. 13, 2020, doi: 10.1109/TSE.2020.2986415.
[92] T. Wolf *et al.*, "HuggingFace's transformers: State-of-the-art natural language processing," 2020.
[93] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.
[94] R. Müller, S. Kornblith, and G. E. Hinton, "When does label smoothing help?" in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 4694–4703.
[95] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, and I. H. Witten, *Weka: A Machine Learning Workbench for Data Mining*. Berlin, Germany: Springer, 2005, pp. 1305–1314. [Online]. Available: http://researchcommons.waikato.ac.nz/handle/10289/1497
[96] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[97] T. T. Le, W. Fu, and J. H. Moore, "Scaling tree-based automated machine learning to biomedical big data with a feature set selector," *Bioinformatics*, vol. 36, no. 1, pp. 250–256, 2020.

Julian Aron Prenner received the MSc degree in computer science from the Free University of Bozen-Bolzano, Bolzano, Italy, where he is also currently working toward the PhD degree. His research interests include automatic program repair, automatic test case generation, and applications of machine learning in software engineering.

Romain Robbes received the master's degree from the University of Caen, Caen, France, and the PhD degree from the University of Lugano, Lugano, Switzerland, in 2008. He is an associate professor with the Free University of Bozen-Bolzano, in the SwSE Research Group. Before that, he was an assistant, then associate professor with the University of Chile. His research interests include empirical software engineering, software evolution, mining software repositories, and machine learning for software engineering.