Few-shot learning for sentence pair classification and its applications in software engineering

Robert Kraig Helmeczi¹, Mucahit Cevik^{1*} and Savas Yildirim¹

¹ Toronto Metropolitan University, 44 Gerrard St E, Toronto,

M5B 1G3, Ontario, Canada.

*Corresponding author(s). E-mail(s): mcevik@torontomu.ca;

Abstract

Few-shot learning—the ability to train models with access to limited data—has become increasingly popular in the natural language processing (NLP) domain, as large language models such as GPT and T0 have been empirically shown to achieve high performance in numerous tasks with access to just a handful of labeled examples. Smaller language models such as BERT and its variants have also been shown to achieve strong performance with just a handful of labeled examples when combined with few-shot learning algorithms like pattern-exploiting training (PET) and SetFit. The focus of this work is to investigate the performance of alternative few-shot learning approaches with BERT-based models. Specifically, vanilla fine-tuning, PET and SetFit are compared for numerous BERTbased checkpoints over an array of training set sizes. To facilitate this investigation, applications of few-shot learning are considered in software engineering. For each task, high-performance techniques and their associated model checkpoints are identified through detailed empirical analysis. Our results establish PET as a strong few-shot learning approach, and our analysis shows that with just a few hundred labeled examples it can achieve performance near that of fine-tuning on full-sized data sets.

Keywords: NLP, Few-shot learning, Transformers, Prompting, Prompt-based learning, PET, Software requirement specification (SRS), Conflict detection

1 Introduction

With the recent development of powerful pre-trained language models such as BERT [7] and GPT-3 [5], many studies have started to investigate the impact of few-shot learning, the practice of training high-quality models using just a handful of labeled training instances. Few-shot learning represents a significant source of savings, both in terms of time and money, for a variety of classification tasks. For instance, in software engineering, detecting bug dependencies and duplicates can be an arduous task for bug triagers. In general, for a repository with n bugs, detecting dependencies and duplicates requires comparing all pairs of bugs, a task which has $\mathcal{O}(n^2)$ complexity. For large n, this can become intractable for even a computer to solve, but even for relatively small values of n, manual comparisons become extremely time consuming. Training a language model to detect these relationships can be an effective approach for automating this procedure, but this generally requires manually labeling tens of thousands of bugs as duplicates or dependent. Labeling such a large number of bug reports is costly, requiring enormous amounts of time and money to be dedicated to this task. For many development projects, even if these costs could be absorbed, there may not be enough bug reports to label. In such problems, training a language model without labeling a substantial amount of training data offers a significant advantage.

With this description of few-shot learning, one quickly notes that it is always preferable to be able to train high-quality models with just a handful of labeled training data. However, few-shot learning might come with a decrease in model accuracy, and the tolerability of such decreases changes with both the task and the overseeing entity. For example, in the automatic essay scoring domain, contract graders are expected to grade with an accuracy between 80% and 95% [4]. Accordingly, it is important to quantify the ability of few-shot learning methods on a variety of tasks to inform the decisions of industry professionals seeking to apply them in their field.

This research focuses on the applications of few-shot learning algorithms in important practical problems in the domain of software engineering. In particular, we focus on pattern-exploiting training (PET) [23] and SetFit [26] as few-shot learning methods, and we compare their performance with vanilla fine-tuning. Much of the literature on few-shot learning generally involves comparing and contrasting various few-shot approaches using different Transformer model checkpoints, or by focusing on just a single checkpoint [26, 25, 13]. This is generally done with the intention of emulating a particular few-shot learning setting, where validation data is not available to evaluate an approach's performance for each checkpoint. Instead, model checkpoints are often chosen based on model performance on full-sized datasets [24], but this approach might not always be indicative of few-shot learning performance. Accordingly, we conduct an empirical analysis with different few-shot learning approaches on a variety of model checkpoints to better inform the selection of the ideal checkpoint.

Another limitation in the current literature is on dataset size. Alex et al. [2] note that, in a few-shot setting, it is generally not desirable to label more than 50 examples for training data, and many studies on few-shot learning consider similarly-small datasets [23, 13, 25]. While labeling more examples may not be preferable, studies that consider only datasets of this size create a huge disparity in the literature between the few-shot learning domain and language modeling on full-size datasets, which often contain thousands or tens-of-thousands of labeled examples. Accordingly, we investigate the performance of few-shot learning approaches beyond this 50-labeled example limit, providing a thorough analysis of each approach over a range of training set sizes. The results provide a comprehensive overview of the leading checkpoints and approaches across various small dataset sizes.

Our contributions to text modeling in software engineering can be summarized as follows:

- Four text classification tasks are introduced as a test suite to evaluate few-shot learning approaches in the software engineering domain. The tasks considered include duplicate detection in bug reports from Bugzilla¹ and questions from Stack Overflow,² bug dependency detection in bugs from Bugzilla, and conflict detection in software requirement specification (SRS) documents. Each of these tasks is formulated as a sentence pair classification problem.
- PET is adapted to each of the considered software engineering applications and is compared with SetFit and fine-tuning to evaluate the performance of few-shot learning in this domain. The ensuing analysis reveals the overall viability of few-shot learning in software engineering applications.
- The few-shot learning approaches are compared using three popular pre-trained language model checkpoints—BERT_{LARGE}, RoBERTa_{LARGE}, and DeBERTa_{LARGE}—as their underlying language model. The numerical analysis reveals that fine-tuning with BERT_{LARGE} is often the best-performing approach when compared to all other checkpoints and approaches in a few-shot setting. The ensuing discussion identifies differences in the training procedures for each model checkpoint which may have led to these observations.
- The performance of each few-shot learning approach is evaluated over a range of dataset sizes. The empirical results identify the strengths of each model checkpoint and few-shot approach with respect to training set size, allowing for more informed decision-making when selecting a few-shot learning approach and the language model.

The rest of this paper is organized as follows. Section 2 reviews the literature on the tasks considered, and then discusses the few-shot learning methods that we use to solve them. Section 3 details the tasks to be solved, the datasets used to represent them, and how we adapt each few-shot learning approach

https://bugzilla.mozilla.org

²https://stackoverflow.com

4 Few-shot learning for sentence pair classification

to these tasks. Section 4 summarizes the results of the numerical experiments for each of the datasets. Section 5 summarizes our findings, identifies limitations and areas of future research, and provides a series of recommendations for applying few-shot learning in a practical setting based on the observations in this work.

2 Literature Review

In this section, we first briefly review the relevant literature on the software engineering classification tasks considered in this study. Then, we discuss the recent advances in few-shot learning methodologies. Table 1 provides an overview of the most relevant studies for the software engineering tasks considered in our work.

Table 1: Summary of recent relevant literature on sentence pair classification problems in software engineering.

Study	Approach	Features	Few-shot?	Task
Aggarwal et al. [1]	Decision-making	C&T Sim.	No	Bugzilla Duplicate
Chauhan et al. [6]	Ranking	TF-IDF vector	No	Bugzilla Duplicate
Kim and Yang [14]	Binary Classification	Topic Modeling	No	Bugzilla Duplicate
Wang et al. [27]	Decision-making	Deep Learning	No	SO Duplicate
Guo et al. [8]	Decision-making	Semantic meta-model	No	SRS
Malik et al. [18]	Decision-making	BERT embeddings	No	SRS
Our study	Decision-making	BERT embeddings	Yes	All Tasks

SO: Stack Overflow, SRS: Software Requirement Specification document conflict detection, All Tasks: Entailment + SRS + Bugzilla Duplicate + SO Duplicate, C&T Sim.: Contextual and Textual Similarity

2.1 Detecting duplicate bug reports

In the context of bug duplicate detection, Lin et al. [16] group solution methods into three main categories. Firstly are ranking approaches which, when given a bug report, return a ranked list of similar matches. Secondly are binary classification approaches, which train a binary classifier to determine if a bug report is a duplicate or not. In their work, Lin et al. [16] found that such classifiers did not offer substantial benefits to triagers as the state-of-the-art still misclassified the majority of duplicates. Thirdly are decision-making approaches, which take pairs of bug reports and determine if the pair are duplicates. Chauhan et al. [6] provide a detailed literature review of the recent literature on duplicate detection in bug reports. Accordingly, we refer the reader to their work for an extensive overview on the field. We summarize here some of the more

recent approaches used for duplicate detection. While in the recent literature, Zhang et al. [28] propose using few-shot learning for bug repairing and Li et al. [15] suggest few-shot learning may be helpful for bug entity extraction and relationship modeling, our review of the literature did not identify any research into few-shot learning for bug duplicate detection. Therefore, we provide a review some of the relevant literature on duplicate detection in a data rich environment.

Chauhan et al. [6] use a ranking approach to identify duplicate bug reports. They use cosine similarity to identify bugs which are similar to one another, choosing a threshold of 0.6, above which bugs are considered to be duplicates. For vectorization, they use vector space modeling where the feature vector is represented as a TF-IDF vector, achieving prediction accuracy of 88.8%. We note that a natural extension of their work would be to apply SBERT [21] for generating strong sentence vectors. Kim and Yang [14] employ BERT to create a binary classification model for duplicate bug detection. They propose using topic modeling for feature extraction, and then using the features for each state of a topic to train a BERT model for determining if a bug report is a duplicate or not. Their results show substantial improvements over the more common approaches in literature (e.g., CNN, LSTM, RandomForest). Aggarwal et al. [1] propose a decision-making model which considers both textual similarity and contextual similarity between bug reports. By including contextual similarity in addition to textual similarity, they find that model performance improved significantly over using textual similarity alone. They use cosine similarity to measure the similarity between features. They construct a feature table with these similarity measures alongside other extracted features and train a decision tree classifier to predict whether or not a pair of bugs are duplicates.

2.2 Detecting duplicate questions on Stack Overflow

We also consider the task of detecting duplicate questions on Stack Overflow in our investigations, providing a more diverse overview of the task of duplicate detection in software engineering. Like Bugzilla, Stack Overflow data is publicly available, making for a popular data set for natural language processing. In a recent study, Wang et al. [27] investigated duplicate detection in Stack Overflow using deep learning. Among the deep learning approaches that they considered are Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM). They consider a decision-making approach as described in the Bugzilla task: that is, they consider pairs of questions and assign a label to each pair. For generating sentence embeddings, they use Word2Vec-based average-pooling [19]. On Stack Overflow, metadata about questions is available, including the language that they involve. Accordingly, Wang et al. [27] considers six different languages for the duplicate detection task. Across all six languages, they find that Word2Vec with LSTM is the best-performing model.

2.3 Detecting conflicting/duplicate requirements

Guo et al. [8] develop Finer Semantic Analysis-based Requirements Conflict detector (FSARC) for detecting conflicts in SRS documents. Requirements are parsed manually into an eight-tuple, forming a semantic meta-model. The specific semantic elements are formulated based on heuristics, and mapping a requirement to the eight-tuple can be an ambiguous task [8]. Despite this limitation, their results show that the algorithm can achieve strong performance for conflict detection. This approach is particularly limited in a few-shot setting as it requires manually mapping each requirement, which is often too time-consuming for few-shot learning [2]. Malik et al. [18] propose the use of transformer models for conflict detection. Their research focuses on a data-rich environment i.e., an environment in which a substantial amount of labeled data is available. We extend their approach to a few-shot setting through the use of fine-tuned models, and also consider few-shot learning specific approaches such as PET and SetFit.

2.4 Bug report entailment

The process of taking in bug reports, labeling them, and assigning them to developers based on severity, ease of fixing, and other features—known as bug triaging—is well-studied in literature. Several recent studies have found that accounting for the dependency between bugs can offer a substantial performance improvement to the triaging process [3, 11, 12]. Specifically, these studies found that by including bug dependencies when assigning reports to developers, the average amount of time it takes to solve a bug is decreased. However, after an extensive literature review, we did not find any models which attempted to train a classifier to detect bug dependencies, neither in a few-shot nor a data rich environment. Given the new-found importance of bug dependencies in the triaging process, we identify the process of training classifiers for dependency detection as a rich environment for new research.

2.5 Few-shot learning methods

Pattern-exploiting training (PET) is an important few-shot learning approach introduced by Schick and Schütze [23]. This approach to few-shot learning relies on access to a large amount of unlabeled data in addition to a small number of labeled examples and works by adding context to a problem through the use of a prompt. Specifically, Schick and Schütze [23] assume that a major hindrance to effective few-shot learning is the fact that it is often difficult to learn the task being solved when access to training data is limited.

As PLMs are trained using an MLM objective, they are already trained to handle the formulations generated by the PET algorithm. As the pattern itself is able to provide additional information beyond the training data alone, it stands to reason that the combined knowledge of the PLM and the additional context added by the pattern can yield better results. However, a single pattern and verbalizer does not always result in improved performance [23, 24]. This

is a result of having no access to a validation set to identify strong-performing patterns and verbalizers [23]. To remedy this, Schick and Schütze [23] propose training an ensemble of models on a set of pattern-verbalizer pairs. Specifically, they create three pattern-verbalizer pairs, and for each pair they train three models using distinct random seeds for a total of nine models.

While training N models on a handful of training instances can be relatively inexpensive for small N (when compared to training a sequence classifier on a full-sized training dataset), both the storage space and the inference time increase N-fold, which is undesirable. Accordingly, Schick and Schütze [23] derive a solution from Hinton et al. [10]'s knowledge distillation approach which leverages access to an abundance of unlabeled data. Specifically, a set of models \mathcal{M} is generated by training each model on a unique pattern-verbalizer pair and random seed where the MLM loss function is given by the crossentropy loss between $\Pr_{(p,v)}(\ell \mid x)$ and the one-hot encoded true label of the training instance, summed over the entire training set [23]. Then, the ensemble is used to generate soft-labels for the examples in the set of unlabeled data. Specifically, for each unlabeled example x, scores are computed via

$$s(\ell \mid x) = \frac{\sum_{m \in \mathcal{M}} w(m) s(\ell || p_m(x), m)}{\sum_{m \in \mathcal{M}} w(m)}$$
(1)

where $s(\ell||p_m(x), m)$ gives the score assigned by the model m to the input $p_m(x), p_m$ is the pattern used for training the model m, and w(m) is the weight associated with the model m. The weight w(m) is equal to the accuracy of the model on the labeled training data before training. This weighting approach favours patterns that perform better even without training [23]. These scores are then converted to a probability distribution via softmax:

$$\Pr_{(p,v)}(\ell \mid x) = \frac{\exp(s(\ell \mid x))}{\sum_{\ell' \in \mathcal{L}} \exp(s(\ell' \mid x))}$$
(2)

Finally, a sequence classifier is trained on the newly generated softly-labeled data and the labeled training data.

SetFit [26] is a recently introduced model which was shown to outperform PET on the RAFT leaderboard, a leaderboard designed to evaluate the performance of few-shot learning models [2]. SetFit relies on sentence transformers to train models. We briefly describe sentence transformers here. As BERT and its variants can handle an input of two sentences, the procedure for finding the most similar pair of sentences in a set is straight-forward, but costly. Specifically, this procedure requires comparing all sentence pairs. For n sentences, the number of comparisons to be made is $\mathcal{O}(n^2)$. Accordingly, Reimers and Gurevych [21] propose Sentence-BERT (SBERT), a modification to BERT and BERT-like models which allows for the generation of strong vectors representing input sequences. Similarity measures such as cosine similarity and clustering can then be used to find similar sentences. These approaches allow SBERT to offer substantial performance improvements for sentence similarity

tasks or sentence representation tasks. SBERT derives a fixed-sized embedding by adding a mean pooling operation to the output of the BERT-like model [21]. While several loss can be used for training an SBERT model, for this research, only cosine similarity loss is used. Specifically, given two sentences, the cosine similarity is computed and mean squared-error loss is employed with this similarity measure.

SetFit [26] uses contrastive learning to substantially increase the number of samples in a few-shot setting. Specifically, given training data consisting of pairs of text and their label (i.e., $\mathcal{T} = \{(x_i, \ell_i)\}$), we can create a binary classification dataset of triplets by taking pairs of examples from the dataset and assigning them the positive label "1" if they share the same class and assigning them the negative label "0" if they have different classes. For each class label, R positive and R negative triplets are generated, and the union of all such triplets are used as the training data for contrastive learning [26].

After the SBERT model is trained on the contrastive dataset, it is used to encode the original training data, thus generating a training set consisting of pairs of word embeddings and their class labels. Tunstall et al. [26] use a logistic regression model as the classification head for this training data. When data is extremely scarce, the training procedure is extremely fast compared to the alternative few-shot learning methods. Additionally, inference time is much quicker for SetFit models. Finally, unlike for PET [23], SetFit does not require manual input nor unlabeled data to train a model: given a training dataset \mathcal{T} , a model can immediately be trained without any task-specific programming.

3 Methodology

There are numerous studies into various text classification problems in the software engineering domain. However, the research into few-shot learning applications to handle these tasks is limited. In this section, we explore the applications of few-shot learning over a handful of software engineering tasks. Specifically, we consider requirement conflict detection, bug duplicate detection, Stack Overflow duplicate question detection and bug entailment/dependency detection.

In the requirement conflict detection task, the objective is to identify conflicts between requirements in a software requirement specification (SRS) document. This is accomplished by generating all possible pairings of requirements and assigning a label to each pairing. The allowed labels are *Neutral*, indicating that the pairing contains unrelated requirements and requires no manual intervention; *Duplicate*, indicating that the pairing contains two (almost) identical requirements that may be simultaneously satisfied; and *Conflict*, indicating that the pairing contains two related requirements which cannot be simultaneously satisfied. A pairing labeled *Conflict* must be addressed manually by revising one or both of the requirements in the pairing. While *Duplicate* requirements do not require manual intervention, they are of interest because if one of the requirements in the pairing is to change, they

could become conflicting. In addition, having *Duplicate* requirements might lead to needlessly repeating some tasks. Accordingly, it is important to consider the pairings labeled *Duplicate* when adjusting requirements.

Bug duplicate detection is an important task in the bug-triaging process for two main reasons. Firstly, in a triaging process where duplicates are not considered, each bug report in a pair of duplicates may be passed to two separate developers, resulting in wasted labour and potentially conflicts in the code created by the bug fixes introduced by each developer. Secondly, a duplicate bug report may contain additional information, i.e., the pairing together can provide a more comprehensive overview of the problem. Considering both reports together is therefore desirable. In the bug duplicate detection task, we consider a collection of bug reports and generate all possible pairings of reports. For each pairing we assign the label *Duplicate*, indicating that the two reports pertain to the same issue, or *Neutral*, indicating that the two reports are unrelated.

In recent literature, bug dependencies have become a popular topic for research in bug triaging [11, 12, 3]. Given two bug reports, u and v, u is said to depend on v if u cannot be resolved without resolving v. Determining dependencies can ensure that v is prioritized so that u can be resolved, but it also offers an additional benefit. It is logical to assign bugs that depend on one another to the same developer, as dependent bugs will often involve the same files [3]. Incorporating these dependencies into the triaging process can therefore reduce the workload on the development team. In the bug entailment task, we consider pairings (u,v) of bug reports. We assign a pairing the label Entailment if u depends on v, and the label Not Entailment if u does not depend on v.

3.1 Datasets

To investigate the viability of few-shot learning methods in software engineering, we consider four tasks:

- 1. detecting conflicts in software requirement specification (SRS) documents,
- 2. detecting duplicate bug reports,
- 3. detecting duplicate Stack Overflow questions, and
- 4. detecting bug dependencies

Note that unlike the generic text classification over individual sentences, each of these tasks involves classifying sentence pairs, (u,v). Table 2 provides summary statistics with respect to word counts for each dataset. The summary statistics in Table 2 are given for the concatenation of u and v, as the sentences are ultimately concatenated for each of the few-shot learning approaches employed. As the number of tokens generated by a sentence depends on the tokenizer—and therefore, the underlying model checkpoint—we consider word counts to give a rough overview of the datasets as a whole. This information is helpful when choosing the maximum sequence length used by the transformer models.

Dataset	Min.	Mean	Median	Max.
SRS Conflict Bugzilla Duplicate Stack Overflow Duplicate Bugzilla Entailment	12 6 6 6	42 ± 17 34 ± 17 28 ± 13 34 ± 17	42 34 28 34	72 70 51 69

Table 2: Word counts for software engineering pair tasks

We consider training sets of size 25, 50, 100, 200, and 400 for all problems. and we use test sets with 2,000 examples and unlabeled datasets for use with PET of size 5,000. For each task we repeat our experiments on three randomly drawn datasets and report average results.

3.1.1 SRS Conflict Detection

The UAV flight range shall be at

least 20 miles from origin.

For the SRS conflict detection task, we make use of a proprietary dataset provided by IBM consisting of pairs of requirement specifications. To each pair, we assign exactly one of three labels: Conflict, indicating that the two requirements cannot be simultaneously satisfied; Duplicate, indicating that while both requirements can be simultaneously satisfied, they are related in some way and if one is changed they may become conflicting; and Neutral, indicating that the requirement pairings are neither in conflict nor are they duplicates. Table 3 provides some examples from the conflict detection dataset to illustrate each of these labels. From this brief overview of the dataset, we note that for the instance labeled *Duplicate*, the words "UAV" and "Hummingbird" are treated synonymously. The language model may have a difficult time noting this distinction with limited access to task-specific training data.

Specification 1 Specification 2 Label The UAV shall instantaneously The Hummingbird shall send the Duplicate transmit information to the Pilot Pilot real-time information about regarding mission-impacting failures. malfunctions that impact the mission The UAV shall only accept com-The UAV shall accept commands Conflict mands from an authenticated Pilot. from any Pilot controller.

The UAV shall be able to transmit

video feed to the Pilot and up to 4 separate UAV Viewer devices at Neutral

Table 3: SRS conflict detection examples.

once.

3.1.2 Bugzilla Duplicate Detection

The first duplicate detection task that we consider is the Bugzilla duplicate detection task, which uses bugs scraped from Mozilla's Bugzilla³ using the REST API. In this task, each sentence pair can be assigned one of two labels: Duplicate, indicating that the two bug reports are duplicates and Neutral, indicating that the bug reports are not duplicates. Table 4 lists the queried fields that we used to construct our Bugzilla-based datasets. For the duplicate detection task, we queried for bugs with a resolution of "DUPLICATE" to find duplicate bug reports. For such bug reports, we queried for the ids listed in the dupe_of field to get the duplicate bug reports. We then joined these results to get duplicate pairs. To create sentence pair data with the Neutral label, we consider open bug reports. As bug resolutions are handled by bug triagers, we collect bugs created no later than December 31st, 2021 to ensure that adequate time was afforded for bugs to be resolved as duplicates. We collect the bugs in reverse-order from this date, ensuring that our data contains the most recent bug reports from this cutoff, and collect bugs from no earlier than January 1st, 2019. When constructing our datasets, we ensure that for all sentence pairs (u,v) in the training dataset, neither u nor v appears in any sentence pair in the test dataset.

Table 4: Bugzilla queried fields.

Field	Description
id summary description creation_time resolution dupe_of depends_on	The primary key identifying this bug report. The title of the bug report which briefly summarizes it. The detailed description of the bug report The time at which the bug report was created. The status of the bug. If blank, the bug is unresolved. A list of bug ids of which this bug is a duplicate. A list of bug ids for other bug reports that this bug depends on.

Table 5 provides a few examples of from this data set.

Table 5: Bugzilla Duplicate Examples

Bug 1	Bug 2	Label
Update preference gets changed to automatically install updates	Settings for "Firefox Update" get lost	Duplicate
Title tooltips flash off immediately	Title-text tooltip for an image appears and then instantly disap- pears again	Duplicate
allocation size overflow when overriding regexp exec to be dumb	Search: Few results are shown with Bing search engine depending on the region	Neutral
Count becomes zero after refreshing the page	Async/await all the things inside PersonalityProviderWorkerClass	Neutral

³https://bugzilla.mozilla.org

⁴https://bugzilla.mozilla.org/rest/bug

3.1.3 Bugzilla Dependency Detection

For the Bugzilla dependency detection task, also called the *entailment* task, we consider a pair of bug reports (u, v) and assign the label Entailment if u depends on v and Not Entailment if it does not. We follow a similar procedure to Section 3.1.2 when constructing datasets for this task. Specifically, we query for bugs over the same time period, we construct Neutral pairs using the same strategy, and we ensure that no training data appears in our test dataset. For this task, we consider the depends_on field from Table 4, following the same procedure as was done for duplicate detection using the dupe_of field. Table 6 provides some examples from the dependency detection dataset.

Table 6: Bug dependency data examples				
Bug 1	Bug 2	Label		
[meta] Oblivious DoH support	Consider to add confirmation mechanism for ODoH	Entailment		
Let the conversations reorder and save the order	Convert conversation list to tree- listbox	Entailment		
Remove boilerplate necessary to introduce system metric media queries.	Browser.getVersion to return 1.3	Not Entailment		
Array.sort is not working properly with return 0	Fix various compile warnings in NSS	Not Entailment		

Table 6: Bug dependency data examples

3.1.4 Stack Overflow Duplicate Detection

Finally, for the Stack Overflow duplicate detection task, we query the Stack Exchange Data Explorer for questions from Stack Overflow.⁵ In this task, as for Bugzilla duplicate detection, we assign each sentence pair one of the labels Duplicate and Neutral. As posts on Stack Overflow generally have a tag indicating the applicable language/concepts, we selected only questions that had the tag Python. As in practice cross-language questions would not be labeled as duplicates, selecting only Python questions would likely be done in a real application of this duplicate detection task. Querying for the Stack Overflow questions involves joining a considerable number of tables. Table 7 provides some example sentence pairs and their labels for this dataset. As for the Bugzilla detection dataset, we consider open posts (i.e., Neutral pairs) from no later than December 31st, 2021 to ensure an adequate amount of time for questions to be closed as duplicates.

For each of the Bugzilla and Stack Overflow-based tasks, a preliminary analysis revealed that using the entire question body had a negative impact on experimental results. As these question bodies are often substantially longer than the maximum sequence length that transformer models are trained on, they generally have to be truncated significantly. For algorithms like PET that introduce a mask token and additional text, the question/bug bodies are truncated even further. Accordingly, for the Bugzilla dependency detection,

⁵https://data.stackexchange.com/stackoverflow/queries

Few-shot learning for sentence pair classification

Title 1	Title 2	Label
IMAP4: How to correctly decode UTF-8 encoded message body?	Python email quoted-printable encoding problem	Duplicate
Jinja2: how to evaluate Python variable in an if-statement?	Jinja2 template evaluate variable as attribute,	Duplicate
python csv read column function not giving correct output	WSGIServer not working with https and python3	Neutral
select and filtered files in directory with enumerating in loop	How to set any polygon the same total width	Neutral

Table 7: Stack Overflow Duplicate Examples

Bugzilla duplicate detection, and Stack Overflow duplicate detection tasks, we consider only the question titles/bug summaries.

3.2 Adaptation of SetFit to software engineering tasks

The contrastive-learning premise for SetFit naturally lends it to classifying individual sentences. However, for our research into the applications of fewshot learning in software engineering, we require models to handle sentence pairs. Accordingly, we extend SetFit to the sentence pair classification task by joining the sentence pair with a separator token, " || ", and applying segment embeddings to each sentence, borrowing the same mechanism which is used for sequence classifier sentence pair classification.

3.3 Adaptation of PET to software engineering tasks

For the software engineering tasks, which all involve sentence pairs, we denote the first sentence by u and the second by v. For the MLM objective, a mask token must be inserted into the pattern. We denote the mask token in patterns by and the barrier between two segments of text in a pattern by " \parallel ".

Bugzilla Bug Dependency Detection Task.

For the Bugzilla bug dependency/entailment task, we consider the three patterns used by Schick and Schütze [23] on their entailment task:

- 1. "v" ? \parallel _____ , "u"

We use the same verbalizer for each pattern:

$$v(\text{Not Entailment}) = \text{No}$$

 $v(\text{Entailment}) = \text{Yes}$

Using common words in our verbalizer as suggested by Schick et al. [22].

Stack Overflow Duplicate Detection Task.

For the Stack Overflow duplicate detection task, we consider the following three patterns:

Springer Nature 2021 LATEX template

14 Few-shot learning for sentence pair classification

- 1. "v"? \parallel _____. "u".
- 2. Are "u" and "v" the same question? _____.
- 3. Are "u" and "v" duplicates? .

We use the same verbalizer for each pattern:

$$v(\text{Neutral}) = \text{No}$$

 $v(\text{Duplicate}) = \text{Yes}$

Bugzilla Duplicate Detection Task.

For the Bugzilla duplicate detection task, we consider the following three patterns:

- 1. "v"? \parallel _____. "u".
- 2. Are "u" and "v" the same problem? _____ .
- 3. Are "u" and "v" duplicates? _____ .

We use the same verbalizer for each pattern:

$$v(\text{Neutral}) = \text{No}$$

 $v(\text{Duplicate}) = \text{Yes}$

Conflict Detection Task

For the Conflict detection task, we use the following three patterns:

- 1. "u"? \parallel _____, "v".
- 2. Given "u", we can conclude that "v" is _____.
- 3. "u" means "v". $\|$ ____.

We use a different verbalizer for each pattern. For Pattern 1, we use

$$v(\text{Neutral}) = \text{Maybe}$$

 $v(\text{Duplicate}) = \text{Yes}$
 $v(\text{Conflict}) = \text{No}$

For Pattern 2, we use

$$v(\text{Neutral}) = \text{neither}$$

 $v(\text{Duplicate}) = \text{true}$
 $v(\text{Conflict}) = \text{false}$

and for Pattern 3, we use

$$v(\text{Neutral}) = \text{Neither}$$

 $v(\text{Duplicate}) = \text{True}$
 $v(\text{Conflict}) = \text{False}$

3.4 Experimental Setup

As outlined by Perez et al. [20], tuning hyperparameters on validation data can have a substantial performance impact on experimental results. Accordingly, we rely heavily on the literature, particularly the works of Schick and Schütze [23], Tam et al. [25], Tunstall et al. [26], and Zhang et al. [29], for choosing hyperparameters, with one exception. In our experimentation, we found that training SetFit models with the same hyperparameters was not robust against a change in model checkpoint. Accordingly, we considered training each SetFit model for 1 and 3 epochs, and found that BERT_{LARGE} and RoBERTa_{LARGE} yielded stronger results when using a single epoch, but DeBERTa_{LARGE} required 3 training epochs. Without this hyperparameter sweep, model performance was degraded substantially.

Aside from this exception, the selected hyperparameters do not vary for each of the algorithms. For few-shot datasets, we use 1,000 training steps for PET and fine-tuning. For all approaches, we employ a batch size of 16, and a learning rate of 10^{-5} , and a maximum sequence length of 256. For training the PET sequence classifier, we use a temperature of 2 for generating soft labels, and train the model for 5,000 steps on the softly labeled data. For full-sized datasets we use a batch size of 16 and train all models for 5,000 training steps using a learning rate of 10^{-5} . We do not conduct hyperparameter tuning on the full-sized datasets.

We implement fine-tuning using the torch-based checkpoints from the transformers library.⁶ We use the implementation of SetFit provided by Tunstall et al. [26]. For PET, we modify the implementation provided by Schick and Schütze [23]⁸ to allow for distributed computing, and introduce our hand-crafted task-specific patterns and verbalizers.

We conduct our experiments using the resources provided by the Digital Research Alliance of Canada (DRAC). For each of the considered approaches—fine-tuning, PET, and SetFit—we use GPUs for improved training and inference speed. For some approaches (particularly those using DeBERTa_{LARGE}), memory usage was high, requiring the usage of GPUs with 32GB of memory. For approaches using BERT_{LARGE}, 16GB was often sufficient. Accordingly, these experiments are not run on identical hardware. In many cases, we also use gradient accumulation over more than one step (while maintaining an effective batch size of 16 as stated above).

4 Results

This section details our results for each of the focused checkpoints on the four tasks. We begin by comparing the performance of each model on full-sized training datasets, and then compare their performance in a few-shot setting.

⁶https://github.com/huggingface/transformers

⁷https://github.com/huggingface/setfit

⁸ https://github.com/timoschick/pet 9 https://alliancecan.ca/

Finally, we investigate their performance with respect to increasing training set size.

4.1 Performance Analysis of Transformer Models

Table 8 provides the accuracy attained by each of the model checkpoints using vanilla fine tuning for 5,000 training steps on the full-sized datasets. Across each task, we observe the DeBERTa_{LARGE} model performing the best, followed by RoBERTa_{LARGE}, followed by BERT_{LARGE}. This is expected as the three models are a set of incremental improvements on BERT_{LARGE}, with the DeBERTa_{LARGE} model being the most recent iteration.

Bugzilla Conflict Entailment Stack Overflow Num. Examples 4,400 3,626 10,949 7.965 92.9 ± 0.4 BERT_{LARGE} 95.1 ± 0.6 87.3 ± 1.3 91.6 ± 0.3 $RoBERTa_{LARGE}$ 95.5 ± 0.2 89.0 ± 1.0 92.8 ± 0.3 94.0 ± 0.6 $\mathbf{96.0} \pm \mathbf{0.2}$ $\mathbf{90.1} \pm \mathbf{1.1}$ $\mathbf{94.6} \pm \mathbf{0.3}$ $\mathbf{96.0} \pm \mathbf{0.2}$ DeBERTa_{LARGE}

Table 8: Full-sized dataset results for each model.

Bold: Best score for a task.

Table 9 provides summary results of few-shot learning using 50 labeled examples. We observe that, unlike the full-sized results, DeBERTa_{LARGE} is no longer the de facto best model, as it only attains the highest accuracy for the SRS conflict detection dataset. Despite performing the worst in the full-sized dataset results, BERT_{LARGE} attains the highest accuracy on two of the four datasets using vanilla fine-tuning, beating out the next best accuracy by 4.6% and 2.9% for the Bugzilla and Entailment datasets, respectively. The results show that, for a fixed model checkpoint, PET tends to be the best performing approach, with the exception of BERT_{LARGE}, where fine-tuning tends to be the best performer. When considering the best performing algorithm for each model (shown with an underline) we see some substantial performance gains: for example, choosing DeBERTa_{LARGE} PET over BERT_{LARGE} fine-tuning for the Bugzilla dupicate detection dataset results in a drop of 10% in accuracy. We also observe that SetFit performs its best for the DeBERTa_{LARGE} model, having considerably lower accuracy for the BERT_{LARGE} and RoBERTa_{LARGE} models when compared to the other methods using the same checkpoint.

4.2 Impact of Training Set Size

In this subsection, we investigate the performance of each few-shot learning approach for the three considered model checkpoints with respect to increasing training set size. We consider training sets of size 25, 50, 100, 200, and 400 and plot their accuracy, weighted average F1 score, and macro average F1 score. We use a \log_2 scale for the x-axis in each of our plots for clarity, meaning that each point on the plot represents a doubling in the amount of training data.

		Bugzilla	Conflict	Entailment	Stack Overflow
$\mathrm{BERT}_{\mathrm{LARGE}}$	Fine-tune PET SetFit	$\frac{90.7 \pm 1.4}{78.9 \pm 4.0}$ 75.8 ± 5.3	75.2 ± 1.1 76.7 ± 1.8 67.7 ± 4.5	$\frac{84.6 \pm 2.8}{76.6 \pm 3.4}$ 70.3 ± 1.6	$\frac{85.8 \pm 1.7}{77.4 \pm 4.8}$ 73.5 ± 3.2
RoBERTa _{LARGE}	Fine-tune PET SetFit	$80.7 \pm 7.6 \\ \underline{86.1 \pm 2.2} \\ 74.3 \pm 3.2$	77.8 ± 2.5 79.7 ± 0.8 68.1 ± 7.5	$74.7 \pm 4.7 \\ \underline{81.7 \pm 2.2} \\ 71.7 \pm 4.2$	75.0 ± 5.7 86.5 ± 3.4 70.1 ± 6.4
DeBERTa _{LARGE}	Fine-tune PET SetFit	$79.4 \pm 4.1 80.7 \pm 1.9 77.9 \pm 5.4$	74.8 ± 4.1 81.3 ± 2.4 78.5 ± 2.9	$74.9 \pm 4.2 \\ \underline{80.1 \pm 1.7} \\ 75.3 \pm 5.0$	80.0 ± 3.0 84.1 ± 8.3 81.4 ± 5.6

Table 9: Few-shot results for 50 labeled examples.

Bold: Best score for a task.
Underline: Best score for a fixed checkpoint on a task.

Figure 1 shows the performance of each algorithm on the Bugzilla dataset. The results for BERT_{LARGE} further illustrate the performance of fine-tuning outlined in Table 9. It is not until 200 data instances that the performance of PET begins to match that of fine-tuning. Notably, fine-tuning with BERT_{LARGE} nears its maximum performance for the considered datasets after just 50 training examples. PET with 100 training instances using Roberta_{Large} or Deberta_{Large} is competitive with fine-tuning using BERT_{LARGE}. We observe that fine-tuning has the largest performance drop when changing model checkpoints, particularly when data access is limited: moving from BERT_{LARGE} to RoBERTa_{LARGE}, the accuracy of fine-tuning decreases by more than 10% when using 25 training instances. In contrast, when switching to RoBERTa_{LARGE}, the accuracy of PET increases by about 5%. These observations and the rest of the results in Figure 1 suggest that while vanilla fine-tuning with BERT_{LARGE} may be a good option for few-shot learning, this checkpoint is not a good base MLM model for PET for this task. Finally, we note that for 25 training instances, SetFit tends to perform similarly to PET but for larger training set sizes, the improved training speed of SetFit is probably not worth the significant decrease in model accuracy.

We next consider the bug dependency detection (entailment) dataset, outlined in Figure 2. Notably, both the Bugzilla duplicate detection dataset and the dependency detection dataset are scraped from Mozilla's Bugzilla bug report repository, but we observe substantially different results between these two datasets. Once again, fine-tuning is the preferred approach when using BERT_{LARGE}, and PET is the superior approach when using the other model checkpoints. Unlike for duplicate detection, we observe that PET with RoBERTa_{LARGE} and PET with DeBERTa_{LARGE} is competitive with classic fine-tuning using BERT_{LARGE}. That is, across all training set sizes, PET either outperforms fine-tuning with BERT_{LARGE} or is within a few percentage points of it. We note that, regardless of the chosen checkpoint, the best SetFit is able to do is to remain competitive with fine-tuning, but it often scores 10% or more

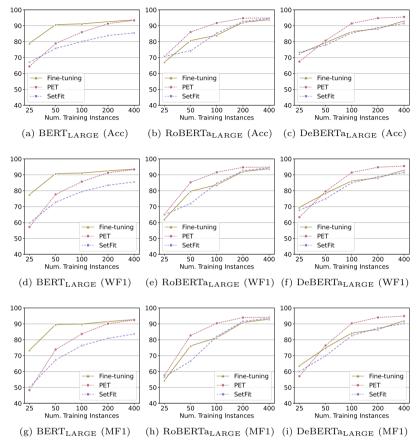


Fig. 1: Bugzilla duplicate detection results.

lower than PET when using the same checkpoint. The difference between PET with Robertalarge and Debertalarge is subtle across most training set sizes, but from BERT_{LARGE} to RoBERTa_{LARGE}, the performance improvements are considerably greater, with a gain of about 5% points in accuracy for 25, 50, and 100 training instances.

Figure 3 gives the results for the Stack Overflow duplicate detection dataset. We observe that for 25 training instances, PET with RoBERTa_{LARGE} is just a few percentage points in accuracy behind fine-tuning with BERT_{LARGE}. However, when we consider the macro average F1-score for these two results, the gap is more than 5%. As early as 50 labeled examples the PET algorithm with RoBERTa_{LARGE} becomes the leader for this task, overtaken by PET with DeBERTa_{LARGE} for the largest dataset sizes. As with our earlier

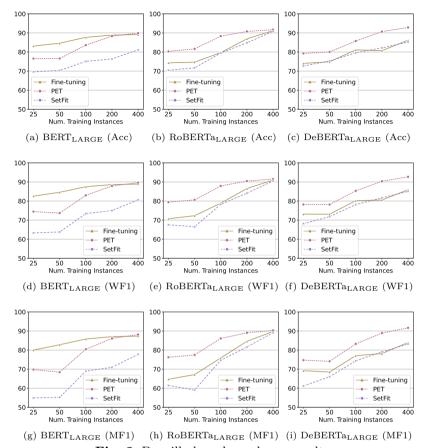


Fig. 2: Bugzilla bug dependency results.

results, we do not observe SetFit to be competitive with the other strategies when using the same checkpoint. However, for smaller training sets, SetFit with DeBERTa_{LARGE} outperforms fine-tuning with RoBERTa_{LARGE} and PET with BERT_{LARGE}. Unfortunately, when compared to PET and fine-tuning with the same underlying language model, SetFit still consistently offers the weakest performance.

Finally, we consider the conflict detection task. Figure 4 outlines the performance of each model on the conflict detection task. This is the only software engineering task that we consider which shows PET with BERT_{LARGE} performing similarly to fine-tuning with BERT_{LARGE}. For this problem, we observe that PET with a DeBERTa_{LARGE}-base tends to be the best performing model across all metrics. We also find that SetFit with DeBERTa_{LARGE}

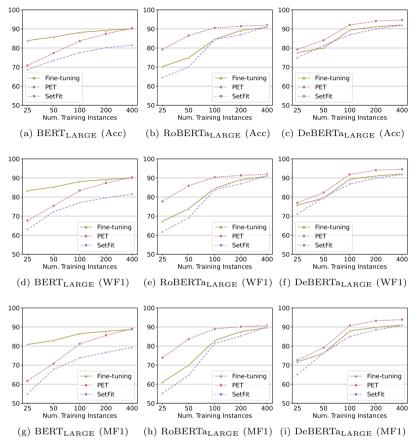


Fig. 3: Stack Overflow duplicate detection results.

outperforms fine-tuning for smaller dataset sizes, and also generally outperforms or matches the performance of the approaches using the other model checkpoints. The results for SetFit are also much more stable here, with mostly a clear pattern of increasing performance for increasing training set size.

5 Discussion and Conclusions

This work focuses on few-shot learning algorithms and their applications in various software engineering tasks. We provide a detailed overview of the current literature, and investigate fine-tuning, as well as the SetFit [26] and pattern-exploiting training (PET) [23] methods. In our analysis, we compare each of these approaches on three transformer model checkpoints: $BERT_{LARGE}$ [7],

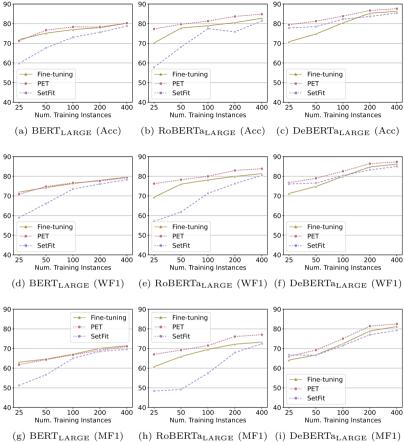


Fig. 4: SRS conflict detection results.

RoBERTa_{LARGE} [17], and DeBERTa_{LARGE} [9]. In the current literature, it is often the case that a single model is selected as the basis for few-shot learning approaches. Accordingly, our investigation provides an overview of the relative performance of each model for each checkpoint. We also measure the robustness of the hyperparameters used for each model, as in a few-shot setting changing additional hyperparameters beyond the model checkpoint is not possible without validation data, which we assume to be unavailable. Our results also investigate the performance of each algorithm for increasing training set size in order to determine the upside of training models with more than just a few training data, but substantially less data than those would be considered as a full-sized data set. Finally, we evaluate each model on a variety of tasks

22

and, in the case of software engineering, we derive these tasks from different problems and data sources to provide a more detailed overview of few-shot learning performance.

In conducting our research, we attempted to curate diverse data sets and problem instances to ensure a thorough investigation of few-shot learning in the software engineering domain. However, our research is not without its limitations. Firstly, we note that when selecting the few-shot learning approaches to experiment with, we consider performance on the RAFT leaderboard [2] to determine the state-of-the-art. A major limitation of the RAFT leaderboard is that it does not contain any data sets which use sentence pairs, meaning that the tasks explored in this study are different from the data used to evaluate models on the RAFT leaderboard. As in the context of few-shot learning there is no development set, the chosen algorithm, checkpoint, and hyperparameters must be chosen based on an educated guess, and leaderboards provide a fair way to inform such decisions. Accordingly, we believe that the next iteration of the RAFT leaderboard can benefit from including sentence pair tasks, as this will allow for better informed decisions in the context of model selection in those cases. Secondly, our model checkpoints are all of a similar architecture. Specifically, Roberta_{LARGE} and DeBerta_{LARGE} are effectively incremental improvements over BERT_{LARGE}. While our results showed that BERT_{LARGE} was still able to outperform these models for some tasks in a few-shot setting, experimenting with a more diverse array of model checkpoints may reveal different relationships between few-shot learning approaches than observed in this work.

We summarize the key takeaways into a list of recommendations. With the understanding that they should be considered together with the limitations of our study, we provide the following recommendations:

- Label as many examples as you can. While it is easy to fixate on which method performs the best when only e.g., 25 examples are available, in practice one will almost always be concerned with maximizing model performance. When we consider a fixed approach and a fixed checkpoint, we observe that the difference in performance between 25 training instances and 400 training instances is frequently 20 absolute percentage points or more for all metrics across most of the problems considered. This represents a substantial improvement and still requires labeling less data than in a full-sized context. Accordingly, if you require the best performance, consider labeling hundreds of examples rather than tens.
- Select the best method for the task you are working on. While leaderboards like Alex et al. [2]'s RAFT provide an "Overall" score to help compare the performance of few-shot learning approaches, the results here show that one cannot consider overall performance alone to draw conclusions. You should look for model performance on datasets similar to the one you are considering. This means look for datasets covering a similar task, with a similar number of labels, which consider the same

sentence classification task (i.e., if you are using sentence pairs, look for a similar dataset using sentence pairs).

• Do not select a model based on performance in a data rich environment. On the SuperGLUE leaderboard, ¹⁰ DeBERTa_{LARGE} is a top-performer. It also performed well on the full-sized datasets. However, we frequently observed that it performed poorly when only 25 training instances were available. As we increased the training set size, we did observe that it became a top performer both for fine-tuning and RoBERTa_{LARGE}, but it is clear that choosing a model based on full-size dataset performance will not yield the best results.

There are many opportunities for future research from this work. Firstly, future research may investigate the use of ensemble methods with fine-tuning and SetFit. In general, this may offer a more fair performance comparison with the PET algorithm, which is ensemble-based. Secondly, we note that we considered only bug summaries and Stack Overflow question titles when handling sentence pair classification. As a single-sentence summary/title is not always available for a given task, an investigation into the performance of few-shot learning approaches for large bodies of text would offer useful insight into this field. Thirdly, as this work focuses on classifying sentence pairs, an investigation into single-sentence classification is necessary for software engineering applications. Fourthly, future research may consider additional models pretrained using NSP loss to evaluate its impact on sentence pair classification tasks.

Disclosure Statement

The authors have no relevant financial or non-financial interests to disclose.

Data Availability Statement

The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

References

- Aggarwal, K., Timbers, F., Rutgers, T., Hindle, A., Stroulia, E., Greiner, R., 2017. Detecting duplicate bug reports with software engineering domain knowledge. Journal of Software: Evolution and Process 29, e1821.
- [2] Alex, N., Lifland, E., Tunstall, L., Thakur, A., Maham, P., Riedel, C., Hine, E., Ashurst, C., Sedille, P., Carlier, A., Noetel, M., Stuhlmüller, A., 2021. Raft: A real-world few-shot text classification benchmark, in: Vanschoren, J., Yeung, S. (Eds.), Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks.

¹⁰https://super.gluebenchmark.com/leaderboard

- URL: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/ file/ca46c1b9512a7a8315fa3c5a946e8265-Paper-round2.pdf.
- [3] Almhana, R., Kessentini, M., 2021. Considering dependencies between bug reports to improve bugs triage. Automated Software Engineering 28, 1-26.
- [4] blees.ai, 2023. blees.ai. https://blees.ai/. Accessed on: April 18, 2023.
- [5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. Advances in neural information processing systems 33, 1877–1901.
- [6] Chauhan, R., Sharma, S., Goval, A., 2023. Denature: duplicate detection and type identification in open source bug repositories. International Journal of System Assurance Engineering and Management, 1–18.
- [7] Devlin, J., Chang, M.W., Lee, K., Toutanova, K., 2019. BERT: Pretraining of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Association for Computational Linguistics, Minneapolis, Minnesota. pp. 4171–4186. https://aclanthology.org/N19-1423, doi:10.18653/v1/N19-1423.
- [8] Guo, W., Zhang, L., Lian, X., 2021. Automatically detecting the conflicts between software requirements based on finer semantic analysis. arXiv preprint arXiv:2103.02255.
- [9] He, P., Liu, X., Gao, J., Chen, W., 2021. Deberta: Decoding-enhanced bert with disentangled attention, in: International Conference on Learning Representations. URL: https://openreview.net/forum?id=XPZIaotutsD.
- [10] Hinton, G., Vinyals, O., Dean, J., 2015. Distilling the knowledge in a neural network. URL: https://arxiv.org/abs/1503.02531, doi:10.48550/ ARXIV.1503.02531.
- [11] Jahanshahi, H., Cevik, M., 2022. S-dabt: Schedule and dependency-aware bug triage in open-source bug tracking systems. Information and Software Technology 151, 107025.
- [12] Jahanshahi, H., Chhabra, K., Cevik, M., Başar, A., 2021. dependency-aware bug triaging method, in: Evaluation and Assessment in Software Engineering, pp. 221–230.
- [13] Karimi Mahabadi, R., Zettlemoyer, L., Henderson, J., Mathias, L., Saeidi, M., Stoyanov, V., Yazdani, M., 2022. Prompt-free and efficient fewshot learning with language models, in: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Dublin, Ireland. pp. 3638–3652. URL: https://aclanthology.org/2022.acl-long.254, doi:10.18653/v1/2022.acl-long.254.
- [14] Kim, T., Yang, G., 2022. Predicting duplicate in bug report using topicbased duplicate learning with fine tuning-based bert algorithm. IEEE Access 10, 129666–129675.

- [15] Li, B., Wei, Y., Sun, X., Bo, L., Chen, D., Tao, C., 2022. Towards the identification of bug entities and relations in bug reports. Automated Software Engineering 29, 24.
- [16] Lin, M.J., Yang, C.Z., Lee, C.Y., Chen, C.C., 2016. Enhancements for duplication detection in bug reports with manifold correlation features. Journal of Systems and Software 121, 223–233.
- [17] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.
- [18] Malik, G., Yıldırım, S., Cevik, M., Bener, A., Parikh, D., 2023. Transfer learning for conflict and duplicate detection in software requirement pairs. arXiv preprint arXiv:2301.03709.
- [19] Mikolov, T., Chen, K., Corrado, G.S., Dean, J., 2013. Efficient estimation of word representations in vector space. URL: http://arxiv.org/abs/1301. 3781.
- [20] Perez, E., Kiela, D., Cho, K., 2021. True few-shot learning with language models. Advances in neural information processing systems 34, 11054– 11070.
- [21] Reimers, N., Gurevych, I., 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks, in: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Association for Computational Linguistics, Hong Kong, China. pp. 3982–3992. URL: https://aclanthology.org/D19-1410, doi:10.18653/v1/D19-1410.
- [22] Schick, T., Schmid, H., Schütze, H., 2020. Automatically identifying words that can serve as labels for few-shot text classification, in: Proceedings of the 28th International Conference on Computational Linguistics, pp. 5569–5578.
- [23] Schick, T., Schütze, H., 2021a. Exploiting cloze-questions for few-shot text classification and natural language inference, in: Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, Association for Computational Linguistics, Online. pp. 255–269. URL: https://aclanthology.org/2021.eacl-main.20, doi:10.18653/v1/2021.eacl-main.20.
- [24] Schick, T., Schütze, H., 2021b. It's not just size that matters: Small language models are also few-shot learners, in: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Association for Computational Linguistics, Online. pp. 2339–2352. URL: https://aclanthology.org/2021.naacl-main.185, doi:10.18653/v1/2021.naacl-main.185.
- [25] Tam, D., R. Menon, R., Bansal, M., Srivastava, S., Raffel, C., 2021. Improving and simplifying pattern exploiting training, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Online and Punta

- Cana, Dominican Republic. pp. 4980–4991. URL: https://aclanthology.org/2021.emnlp-main.407, doi:10.18653/v1/2021.emnlp-main.407.
- [26] Tunstall, L., Reimers, N., Jo, U.E.S., Bates, L., Korat, D., Wasserblat, M., Pereg, O., 2022. Efficient few-shot learning without prompts. arXiv preprint arXiv:2209.11055.
- [27] Wang, L., Zhang, L., Jiang, J., 2020. Duplicate question detection with deep learning in stack overflow. IEEE Access 8, 25964–25975. doi:10.1109/ACCESS.2020.2968391.
- [28] Zhang, J., Cambronero, J., Gulwani, S., Le, V., Piskac, R., Soares, G., Verbruggen, G., 2022. Repairing bugs in python assignments using large language models. arXiv preprint arXiv:2209.14876.
- [29] Zhang, T., Wu, F., Katiyar, A., Weinberger, K.Q., Artzi, Y., 2021. Revisiting few-sample {bert} fine-tuning, in: International Conference on Learning Representations. URL: https://openreview.net/forum?id= cO1IH43yUF.

A Alternative few-shot learning approaches

PET is a highly popular algorithm in the literature. As of today, the combined number of citations on Google Scholar¹¹ for Schick and Schütze [23] and Schick and Schütze [24] exceeds 1000. Naturally, many authors have investigated methods for improving upon the performance of the PET algorithm. We discuss a few of these variants in the literature here and why we chose to use PET over these variants.

Firstly, we review Iterative PET (iPET), which was introduced in the original PET paper [23]. iPET involves training generations of PET models on increasingly larger training sets. Specifically, the training data is used to train a set of masked language models \mathcal{M} . Then, for each model $m \in \mathcal{M}$, a new training set is created by combining the original labeled training data with a unique portion of the unlabeled data. This unlabeled data is given soft labels by a subset of the models in \mathcal{M} . Accordingly, after the first generation, each model m is trained on a different set of training data. At each iteration, the amount of unlabeled data used is increased and this process is repeated for several iterations. Finally, the complete unlabeled dataset is given soft labels, and a sequence classifier is trained on the union of the labeled training set and the softly-labeled unlabeled dataset as in traditional PET. Schick and Schütze [23] reported that iPET performed better than PET in all cases. The empirical results in the literature often show that iPET only offers a small performance improvement at the cost of significantly increased training time. Accordingly, we do not investigate its use here.

Tam et al. [25] propose ADAPET, an extension of PET which does not require access to unlabeled data. The foundation for ADAPET is a redesigned loss function which consists of two components: decoupled label loss and label-conditioned MLM loss. The decoupled label loss modifies how the algorithm

¹¹https://scholar.google.ca

treats tokens that do not belong to the subset of tokens in the verbalizer. Specifically, in the PET algorithm, the score for a given label $s(\ell || p_m(x), m)$ is calculated by considering only those tokens represented in the verbalizer. That is, only tokens t for which there exists a label ℓ such that $t \in v(\ell)$ contribute to the gradient. In ADAPET's decoupled label loss, tokens not belonging to the subspace defined by the verbalizer are also considered in the loss calculations. This is accomplished by computing a softmax probability normalized against the entire vocabulary. Then, the probability of the correct tokens are maximized and the probability of the incorrect tokens (i.e., those tokens from $v_{\ell' \in \mathcal{L}, \ell' \neq \ell}$) is minimized.

The label-conditioned MLM loss of ADAPET involves further MLM training. Specifically, given an example x whose label is ℓ_x , tokens from x are masked to give x'. For each label ℓ , the model is trained to predict the correct tokens from x' if $\ell = \ell_x$, and it is trained to *not* predict the correct tokens from x' if $\ell \neq \ell_x$. The decoupled label loss and the label-conditioned MLM loss are summed together to form the loss for ADAPET.

A key difference between Schick and Schütze [23]'s PET algorithm and Tam et al. [25]'s ADAPET algorithm is that the former assumes access to a large set of unlabeled text data, whereas the latter assumes access to a large set of labeled text data as a development set. Perez et al. [20] investigate the performance of ADAPET without access to a labeled development set, and show that its performance is worse than that of PET. To ensure a true few-shot learning setting, we assume no access to a development set and, accordingly, choose to use the PET algorithm as, in this context, it is the state-of-the-art.

Karimi Mahabadi et al. [13] develop a Prompt-Free and Efficient paRadigm for FEw-shot Cloze-based fine-Tuning (PERFECT), an alternative to PET which does not require hand-crafting patterns and verbalizers. The PERFECT algorithm introduces task-specific adapters, which are trainable additional components added to the model architecture. Specifically, the underlying PLM has its weights frozen, as training an entire model can be unstable and sampleinefficient when access to data is limited. Karimi Mahabadi et al. [13] also freeze the embedding layer and add a separate embedding for labels, where they fix the number of tokens allotted to each label. Mask tokens are appended to the input sentence and the model is trained to predict the label in these mask positions. In particular, the model is trained to predict the correct label by optimizing the label embeddings: notably, given the fixed number of tokens per label, a classifier is trained for each token and multi-class hinge loss is used over the mask positions. As with all few-shot learning approaches, evaluating model performance or handling random variations is important for developing a good model. For PET, this is achieved through the use of unlabeled data. For PERFECT, a validation set is used to train a model. However, unlike for ADAPET, Karimi Mahabadi et al. [13] use half of the limited training data as their validation set: i.e., given 32 labeled instances, they set aside 16 for training and 16 for validation. PERFECT is shown to perform worse than PET on the RAFT leaderboard [2]. Accordingly, we do not experiment with it.

B Stack Overflow data queries

For several tasks, we query for data from open-source databases. For data scraped from Stack Overflow, the queries are SQL-based. We query the Stack Exchange Data Explorer.¹² As these queries are non-trivial, we provide them below in lieu of describing them in detail.

To generate pairings labeled *Neutral*, we query for open posts that have been posed no later than December 31st, 2021. We only consider open questions that have been answered at least once, as an answer ensures that the question received adequate attention and therefore is more likely to have been closed as a duplicate if it was one. We consider only questions tagged *python*, as similar questions using different languages are not generally considered duplicates.

Listing 1 Query for *Neutral* questions

```
SELECT
      TOP 10000 p.id,
      p.title,
      p.creationdate,
      p.body
FR.OM
    Posts AS p
WHERE
      P.closeddate IS NULL
      /* use old posts (eg., before 2022)
       * to ensure enough time for
       * duplicates to be closed */
      AND p.creationdate < '20220101'
      /* only select questions that
       * have been answered (more likely
       * to be understood), but you might
       * want to drop this */
      AND p.answercount > 0
      AND p.tags LIKE '%python%'
ORDER BY
    p.creationDate DESC;
```

As for *Neutral* questions, we consider *Duplicate* questions tagged python.

C Replication study with other datasets

Tunstall et al. [26] report the performance of SetFit with an MPNet base versus fine-tuning (among other algorithms) in their work by experimenting with

¹²https://data.stackexchange.com/stackoverflow/queries

Listing 2 Query for *Duplicate* questions

```
SELECT
    TOP 10000 p.id,
    p.title,
    p.creationdate,
    p.tags,
    p.body,
    p.closeddate.
    c.name AS CloseReason,
    d.id AS dupid,
    d.creationdate AS dupcreationdate,
    d.title AS duptitle,
    d.body AS dupbody
FR.OM
    Posts AS p
    JOIN PostLinks AS pl ON P.id = pl.postid
    JOIN Posts AS d -- for duplicates
    ON pl.relatedpostid = d.id
    JOIN PostHistory AS h ON p.id = h.postid
    JOIN CloseReasonTypes AS C ON h.comment = c.id
    /* if postHistoryTypeId = 10 (see below)
     * then h.comment points to the ID of
     * the close reason */
WHERE
    p.PostTypeId = 1 -- is a question
    AND pl.linktypeid = 3 -- is a duplicate
    AND p.deletiondate IS NULL -- was never deleted
    AND h.PostHistoryTypeId = 10
    AND p.tags LIKE '%python%' -- tagged "python"
ORDER BY
    p.creationDate DESC;
```

publicly available data sets. As our results found that SetFit often underperformed or matched the performance of fine-tuning, we attempt to replicate the performance improvement of SetFit observed by Tunstall et al. [26].

C.1 Datasets

We consider three publicly available datasets from [26]: SST-5,¹³ AmazonCF,¹⁴ and Emotion.¹⁵ For each dataset, two training set sizes are considered, constructed using a balanced sampling of 8 shots/label and 64 shots/label,

¹³https://huggingface.co/datasets/SetFit/sst5

 $^{^{14}}$ https://huggingface.co/datasets/SetFit/amazon_counterfactual_en 15 https://huggingface.co/datasets/dair-ai/emotion

30

respectively. We evaluate all models on the publicly available test split of each dataset. To ensure that the results are not skewed by an unfavourable selection of training data, for each dataset size we sample training data three times, training a model on each sample, and report the average and standard deviation of the performance.

C.2 Experimental Setup

As in Tunstall et al. [26], we consider SetFit with an MPNet base, denoted SetFitmpnet, and fine-tuning with Robertalard. We consider two realizations of the fine-tuning procedure. Firstly, we consider the fixed-hyperparameter fine-tuning considered throughout this work, in which the checkpoint is fine-tuned for 1,000 training steps with a batch size of 16. We denote this as FineTune. Then, we consider the procedure used in [26]. In this case, the few-shot training data is split into an 80–20 train-validation split. The number of training epochs is chosen inclusively between 25 and 75 as the best performer on the validation split, averaged over ten runs (to account for the impact of random starts). This fine-tuning procedure uses a batch size of 4. We denote this as FineTunesettly.

It is worth noting that there are several variations of the different models in the literature for SBERT and SetFit in general. Tunstall et al. [26] use a variation of RoBERTa_{LARGE} which is pre-trained further on sentence pairs to achieve improved performance. In our experiments, we found that this model did not provide uniform improvement over the traditional RoBERTa_{LARGE} model. As such further pre-trained models do not exist for BERT_{LARGE} nor DeBERTa_{LARGE}, we opted to keep a consistent model architecture by simply using mean pooling to construct the sentence transformer models used for our experiments.

C.3 Numerical Results

Table 10 shows the performance of each model on the considered datasets. For 8 shots per label, we observe that both the FINETUNE_{SETFIT} and the SETFIT_{MPNET} values are all within error of the results observed in [26]. More importantly, we observe that SetFit offers a substantial performance boost in this context, including over FINETUNE (i.e., the fine-tuning approach employed in our work). For 64 shots per label, we observe that our results are within error for both the Emotion and the SST-5 datasets, where in each case SETFIT_{MPNET} outperforms FINETUNE_{SETFIT}. We do observe that the AmazonCF results for ||N|| = 64 differ slightly from those in [26]. However, as we are using different test sets and averaging over three different training data samples, it is not necessarily the case that our results will match exactly. Aside from this one outlier, we have largely reproduced the results observed in [26], notably its performance improvement over fine-tuning in a few-shot setting.

¹⁶https://huggingface.co/sentence-transformers/all-roberta-large-v1

Table 10: Replication of SetFit versus Fine-tuning results.

	AmazonCF	Emotion	SST-5		
$ N = 8^*$					
$\mathrm{FineTune}_{\mathrm{SetFit}}$	12.6 ± 4.5	27.8 ± 4.5	33.8 ± 0.8		
FINETUNE	24.0 ± 7.6	40.6 ± 3.3	33.6 ± 2.1		
${\rm SetFit}_{{\rm MPNet}}$	$\textbf{45.6} \pm \textbf{17.1}$	49.8 ± 5.1	44.4 ± 0.8		
$ N = 64^*$					
$\mathrm{FineTune}_{\mathrm{SetFit}}$	$\textbf{70.2} \pm \textbf{3.5}$	71.2 ± 3.5	50.0 ± 1.3		
FINETUNE	63.3 ± 5.9	71.2 ± 3.8	49.7 ± 1.2		
${\rm SetFit}_{{\rm MPNet}}$	65.8 ± 2.5	74.5 ± 2.9	52.4 ± 1.4		

^{*} ||N|| indicates the number of shots per label.

Reported values are MCC for AmazonCF, Accuracy for Emotion and SST-5 ${\bf Bold:}$ Best score for a task.