Information and Software Technology 135 (2021) 106566

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model

Lobna Ghadhab [a], Ilyes Jenhani [b], Mohamed Wiem Mkaouer [c], Montassar Ben Messaoud [a,*]

[a] LARODEC, ISG Tunis, Le Bardo, Tunisia
[b] Department of Information Technology, College of Computer Engineering and Science, Prince Mohammad Bin Fahd University, Al-Khobar, 31952, Kingdom of Saudi Arabia
[c] Rochester Institute of Technology, Rochester, NY, USA

ARTICLE INFO

Keywords:
Software maintenance
Commit classification
Code changes
Deep neural networks
Pre-trained neural language model

ABSTRACT

Context: Analyzing software maintenance activities is very helpful in ensuring cost-effective evolution and development activities. The categorization of commits into maintenance tasks supports practitioners in making decisions about resource allocation and managing technical debt.
Objective: In this paper, we propose to use a pre-trained language neural model, namely BERT (Bidirectional Encoder Representations from Transformers) for the classification of commits into three categories of maintenance tasks — corrective, perfective and adaptive. The proposed commit classification approach will help the classifier better understand the context of each word in the commit message.
Methods: We built a balanced dataset of 1793 labeled commits that we collected from publicly available datasets. We used several popular code change distillers to extract fine-grained code changes that we have incorporated into our dataset as additional features to BERT's word representation features. In our study, a deep neural network (DNN) classifier has been used as an additional layer to fine-tune the BERT model on the task of commit classification. Several models have been evaluated to come up with a deep analysis of the impact of code changes on the classification performance of each commit category.
Results and conclusions: Experimental results have shown that the DNN model trained on BERT's word representations and Fixminer code changes (DNN@BERT+Fix_cc) provided the best performance and achieved 79.66% accuracy and a macro-average f1 score of 0.8. Comparison with the state-of-the-art model that combines keywords and code changes (RF@KW+CD_cc) has shown that our model achieved approximately 8% improvement in accuracy. Results have also shown that a DNN model using only BERT's word representation features achieved an improvement of 5% in accuracy compared to the RF@KW+CD_cc model.

1. Introduction

Software evolution, since being coined by Lehman in 1969, has been the plateau of analyzing the complex nature of how software evolve. Maintenance cost estimation, defect resolution scheduling and technical debt management represent a few of several tasks that drive the high cost of software maintenance, which can reach up to 90%, in comparison with the remaining software development life cycle phases.

Many studies focusing on the analysis of software maintenance activities [1–4] have been carried to closely monitor the software evolution phase and to provide managers with insights on how to effectively manage costs and plan development tasks. In this context, several studies [2,5–9] have focused on categorizing code changes into 3 main maintenance categories: (1) *Corrective*, e.g., fixing errors and faults observed during software use, (2) *Perfective*, e.g., improving software quality attributes such as performance, maintainability, usability, etc. and (3) *Adaptive*, e.g., adapt the software to new environment (software, hardware, etc.) or add new functionalities, etc. Such categorization, supports practitioners in making decisions with respect to various aspects such as resource allocation, choice of frameworks and technologies, managing technical debt, etc. Technically, source code changes are being profiled by their corresponding commit messages, which represent the developer's explanations of the performed change in the code. Commit message classification was the subject of several studies [2–4,10]. These studies have been advancing the mining of source code to better characterize the underlying changes.

* Corresponding author.
E-mail addresses: lobnaghadhab@gmail.com (L. Ghadhab), ijenhani@pmu.edu.sa (I. Jenhani), mwmvse@rit.edu (M.W. Mkaouer), montassar.benmassaoud@isgs.u-sousse.tn (M. Ben Messaoud).

https://doi.org/10.1016/j.infsof.2021.106566
Received 7 May 2020; Received in revised form 25 February 2021; Accepted 28 February 2021
Available online 10 March 2021

For instance, several studies have analyzed software commits, to extract fine-grained change patterns, such as refactoring operations [11,12], API migrations [13–15] and bug fixes [16].

To extract useful information from the commit messages, existing studies used keywords extracted from the commit messages. These keywords have been used along with source code changes in the classification of commits. Using static keywords or any other embedding method that does not take into account the context of each word in the commit message will lead to incorrect classification of ambiguous commit messages, i.e., messages that contain words that can be used in commit messages that belong to different maintenance categories.

To overcome this limitation, we propose to use Bidirectional Encoder Representations from Transformers (BERT) [17], a pre-trained neural language model that provided state-of-the-art results in many natural language processing tasks including text classification. With the help of transformers, the model can better determine the context of each word in the commit messages.

We also aim at conducting a deep analysis of the impact of the introduction of different source code changes on the overall performance of commit classifiers as well as for each maintenance category. Specifically, we intend to investigate how source code changes support commit classification when using transformer encoders for the commit messages. We will determine which source code changes represent a reliable source for the classification of commits into each one of the adopted maintenance categories (corrective, perfective and adaptive) by combining BERT word representations of the commit message and code changes extracted from the same commit.

We started with creating a labeled dataset of 1793 commits that we have collected from publicly available datasets and that we have made available to the public.[1] For each commit, we used popular code change mining tools to extract a fine-grained set of source code changes that we have included as additional features to the set of features we obtained using the BERT model. Different versions of the dataset have been then used to train and test deep neural network (DNN) models that will be used later on for the classification of unseen commits.

The main contributions of this work can be summarized in what follows:

- The use of BERT to encode commit messages which will provide the commit classifier with a better understanding of each word in the message. This will improve the performance of the commit classification model which is in turn based on a fully connected deep neural network (DNN).
- The deep experimental analysis of the impact of introducing different categories of source code changes on the overall as well as the per-category commit classification performance.
- The collection of a balanced benchmark dataset of manually annotated commits. For each commit, an extended set of fine-grained source code changes is also provided.

The rest of the paper is organized as follows: Section 2 provides a survey of previous works on commit classification. Section 3 provides an overview of maintenance and code change categories. This section also introduces the concept of transfer learning in natural language processing and the BERT pre-trained language model. Section 4 presents a description of our proposed commit classification approach. Experiments are presented and discussed in Section 5. Threats to the validity of this work are discussed in Section 6. Finally, Section 7 concludes the paper.

2. Related work

Understanding and organizing software changes is becoming more and more challenging especially with complicated projects that go through a lot of changes during their development process. In this context, researchers have proposed different methods to improve software quality dealing with commit classification and software change prediction.

2.1. Commit classification

Several studies have been conducted on automatically classifying software changes using commit messages. Hindle et al. [4] used machine learning techniques to automatically classify large changes into the maintenance categories proposed by Swanson [18] (corrective, adaptive and perfective), and included feature addition and non-functional changes as additional categories. To describe a commit, they used word distribution of the commit message, its author and the modified modules. Gharbi et al. [3] have also proposed a commit classification method using a multi-label active learning approach. In their approach, a commit can be labeled by more than one of the Swanson's maintenance categories at a time. The active learning approach was helpful in obtaining good classification performance without the need of labeling all available commits. Levin et al. [19] proposed new metrics that capture temporal and semantic developer-level information collected on a per developer basis to classify commits into the same aforementioned maintenance categories. Mockus et al. [10] proposed a word frequency analysis with normalization to select keywords from the commit messages and used them as features. Amor et al. [20] adopted a Naive Bayes model to classify code transactions (atomic commits which modify at least one source code file) into 4 activities (corrective, perfective, adaptive and administrative) based on their textual descriptions using bag of words. Zafar et al. [21] proposed a binary commit classification approach where each commit is classified as either "bug fix" or not. Instead of using keywords, the authors have used the Bidirectional Encoder Representations from Transformers (BERT) [17] that can understand the context of the words in the commit messages. In this work, we are also using the BERT model but in an extended context of multi-class classification where each commit will be classified into one of the 3 Swanson's maintenance categories (corrective, adaptive and perfective).

2.2. Source code changes usage

In addition to the aforementioned approaches which mainly used features extracted from commit messages and/or commit metadata, other approaches have introduced additional features extracted from the source code. Levin et al. [2] proposed a method to automatically classify software changes (commits) using both source code changes and a pre-defined list of keywords extracted from the commit messages. Similarly, Mariano et al. [1] improved commit classification accuracy by using the XGBoost classifier and by introducing three additional features to the set of features used in [2], namely, the total added LOC (lines of code), total deleted LOC and the number of files changed per commit.

Source code changes have been also used to solve other software engineering related problems. For instance, Zhou et al. [9] used source code changes to automatically detect security issues and find unidentified vulnerabilities in open source libraries by using natural language processing and machine learning techniques.

In another problem, Knyazev [7] proposed an automatic classification method of code changes to optimize code review and code change control on final development stages. These code changes have been categorized into different low-level operations such as new functionality, bug fix, refactoring, cosmetic changes and code deletion. Weißgerber et al. [8] proposed an approach to identify refactoring operations from source code changes. Chakraborty et al. [5] proposed a novel Tree2Tree Neural Machine Translation system to model source code changes and learn code change patterns from the wild. Herzig et al. [6] used source code change genealogy graphs to define a set of change genealogy

[1] https://zenodo.org/record/4266643#.X6vERuLPxPY.

Table 1
Characteristics of commit classification studies.

Study	Year	Uses code changes	Classification method	Category	Training size	Result
[22]	2008	NO	Systematic labeling	Swanson's category Feature addition Non-Functional	2000 commits	Not reported
[4]	2009	NO	Machine learning classifiers	Swanson's category Feature addition Non-Functional	2000 commits	F-Measure: 50%
[23]	2013	NO	Systematic labeling	Design improvement Extension Backward compatibility Abstraction level refinement	Not mentioned	Not reported
[24]	2015	NO	Manual labeling	Swanson's category [25] category Non-Functional	967 commits	Not reported
[26]	2016	NO	Topic modeling	Swanson's category	80 commits	F-Measure: 76%
[2]	2017	Yes	Machine learning classifiers	Swanson's category	1151 commits	Accuracy: 73%
[21]	2019	No	BERT binary classifier	Bug Fix	2000 commits	Accuracy: 85%

network metrics that describe dependencies of change sets that can be used to identify bug fixing change sets without using commit messages and bug reports. Loyola et al. [27] proposed a model to automatically describe the source code changes of a given program based on a set of commits, which contains both the modifications and the message introduced by the user.

What distinguishes our work from the existing works, especially from Levin et al.'s approach [2], is that we aim at conducting a deeper analysis of the impact of an extended set of code changes extracted by using different change distiller tools (Code distillery, Refactoring-miner and Fix-miner) on each of the 3 commit categories (corrective, perfective and adaptive) and propose a guideline on when to use each one of these tools during the classification of commits. We also aim at verifying if code changes still boost commit classification when using state-of-the-art contextual pre-trained language models for text classification such as BERT.

Table 1 summarizes the related works.

3. Background

In this section, we start by providing short and simple definitions of the maintenance categories proposed by Swanson [18] and required in this work.

We then provide an introduction to transfer learning in the context of Natural Language Processing (NLP) and present the state-of-the-art transfer learning model: Bidirectional Encoder Representations from Transformers (BERT), a pre-trained neural language model that revolutionized many NLP tasks including text classification.

3.1. Commit categories

3.1.1. Corrective

This is a corrective activity that includes fixing all types of possible problems (e.g., bugs, faults, defects, etc.). A bug is an error in the program, that causes the program to behave not as it was intended by the developer. Bugs can be classified into major or minor bugs according to their severity and their impact on the system.

3.1.2. Adaptive

It is a functional activity where changes are made to adapt the software to new environments, platforms and software dependencies. Sometimes, adaptive changes reflect organizational policies or rules as well.

Table 2
Real examples of commit messages belonging to the categories considered in this study.

Category	Commit message	Project
Corrective	"Fix Completable swallows-OnErrorNotImplementedException"	ReactiveX-RxJava
Perfective	"Performance improvement for Element.text"	jhy-jsoup
Adaptive	"Implement a cached thread scheduler using event"	ReactiveX-RxJava

3.1.3. Perfective

This activity refers to changes that do not reflect the functional behavior of a software. It includes design, performance improvements and other non functional requirements. These activities aim at improving program structure (e.g., renaming or moving entities, pushing down attributes, code clean up, etc.) and system's efficiency (e.g., improving algorithms, choosing more efficient data structures or library routines, and operating system tuning).

Table 2 showcases examples of commit messages that belong to each one of the aforementioned categories. These commit messages were extracted from the dataset we used in our study.

3.2. Transfer learning for Natural Language Processing (NLP)

In traditional learning, when we design a machine learning-based solution for a specific problem, we need to use a dataset related to that specific task, then train (fit the parameters of) the machine learning model from scratch. When we tackle a different problem, we usually start again from scratch, gather another dataset and retrain the model again. However, we humans, use what we have learned from all the previous tasks and problems we have faced in order to solve new unseen problems. That is exactly the main idea behind transfer learning.

Transfer learning is an approach in which the knowledge learned from a large-scale dataset to solve a particular task is reused (transferred) and applied to solve a different but related task. Technically speaking, transfer learning helps us use pre-trained machine learning and deep learning models and fine-tune them using a relatively small labeled dataset from the downstream tasks. Transfer learning has shown state-of-the-art results in many NLP-related problems (e.g., text classification, part of speech tagging, question answering, text translation, etc.) since all these problems share a common element: knowledge about the language and this is what made transfer learning a successful solution for many supervised NLP tasks.

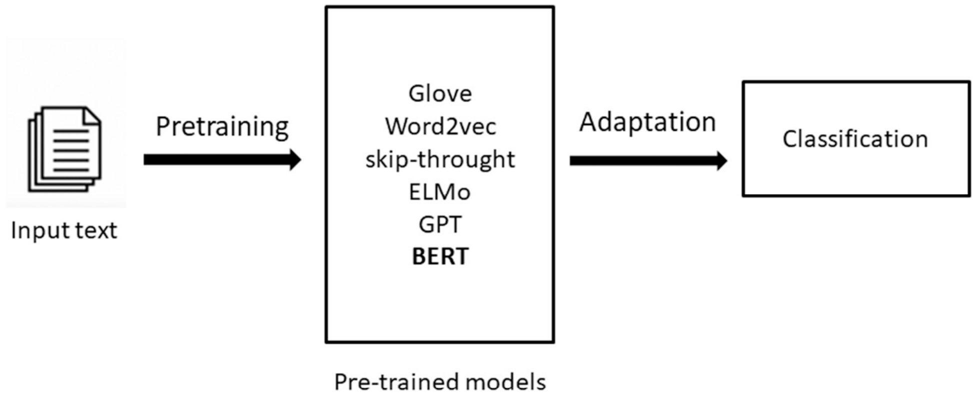

Fig. 1. Transfer learning for NLP.

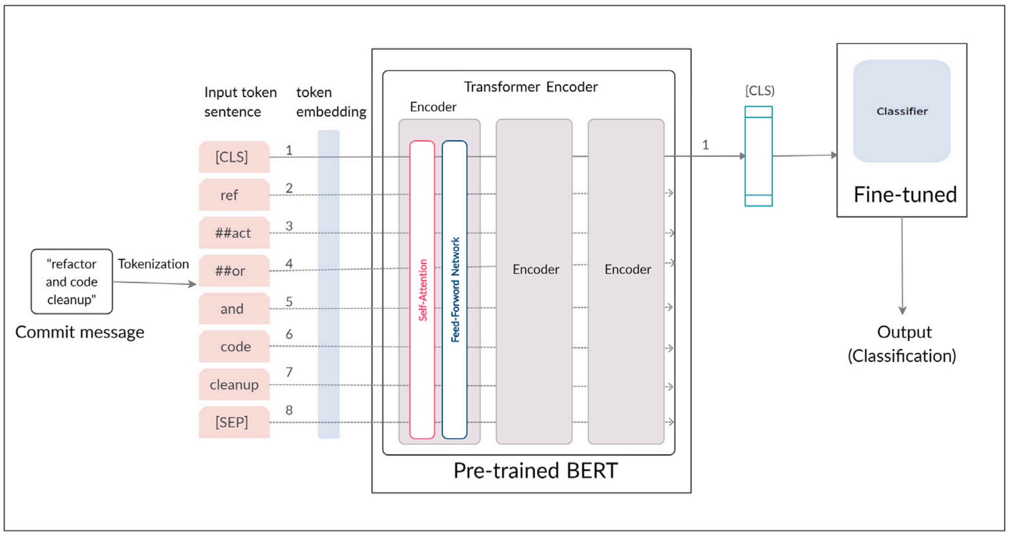

Fig. 2. Fine tuning BERT for commit classification.

Our commit classification approach is based on transfer learning where the output of a pre-trained BERT model is used as input for a separate deep neural network for text classification.

As shown in Fig. 1, input text is fed to a pre-trained model (e.g. BERT) then the BERT's output, i.e., the pre-trained representations, are used as input to train another classifier.

3.3. Bidirectional Encoder Representations from Transformers (BERT)

BERT [17] is a transformer-based language model that is pre-trained on a huge dataset built from the BookCorpus and the English Wikipedia. It was proposed by Google Research and has given state-of-the-art results in several NLP problems.

Unlike context-free models (e.g. Word2vec and GloVe) which generate a single word embedding representation for each word in the vocabulary, transformer-based models [28] take into account the context for each occurrence of each word. To achieve this, transformer models which have been proposed to handle sequence-to-sequence tasks (e.g. machine translation, text-to-speech transformation, etc.) introduced the concept of *Attention* and use an Encoder–Decoder architecture. The *Attention* mechanism helps these models determine how much attention to pay to each input word.

BERT was initially released in two sizes BERT-base (a neural network model with 12 layers, 768 hidden units, 12 attention heads and 110M parameters (weights)) and BERT-large (24 layers, 1024 hidden units, 16 attention heads and 340M parameters). In this work, we used DistilBERT [29], a light version of BERT-base that has 6 layers of encoders stacked on top of each others, 768 hidden units, 12 attention heads and 66M parameters. Compared to BERT-base, DistilBERT has 40% less parameters and runs 60% faster while preserving good performances. Since we are using BERT for text classification,

A BERT pre-trained model takes as input a sequence of words (a sentence) that it tokenizes into a maximum of 512 tokens using its vocabulary. The IDs of these tokens will pass through the different encoder layers of BERT to produce the words representations as vectors of size 768 each.

In this work, we used BERT for commit classification by adding on top of the final BERT's layer, another classification layer that uses BERT's output as input dataset. This way, our approach will benefit from the rich knowledge learned by BERT during its pre-training process.

As shown in Fig. 2, BERT is basically a pre-trained transformer encoder stack. An input message (e.g. a commit message) first undergoes a pre-processing step. This step includes: tokenization and padding. Each commit message is split into a sequence of tokens using BERT tokenizer which uses a large vocabulary. Then two specific tokens will be added to these input tokens, namely, [CLS] and [SEP]. [CLS] is the special classification token and [SEP] represents the separation token that is used to separate two input sentences for sequence-to-sequence problems. In our case, since we are dealing with a classification problem which only requires one input sentence, [SEP] is used to indicate the end of the sentence. Since commit messages will have different numbers of tokens, another special token [PAD] will be added after the [SEP]

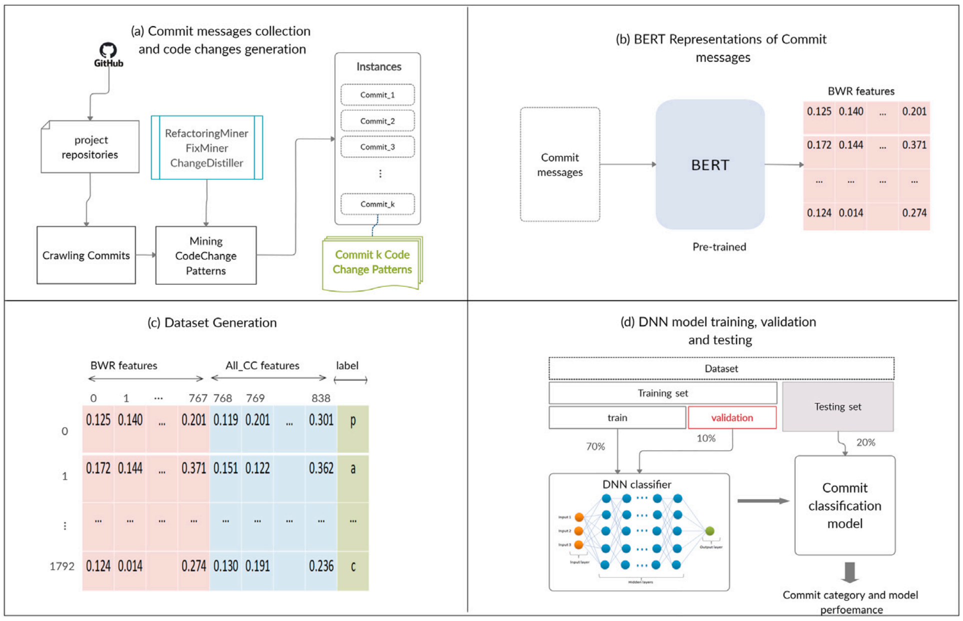

Fig. 3. Process of commit classification using DNN, BERT and code changes.

tag to make all BERT input equally sized. All input tokens will be then replaced by their IDs which constitute the token embeddings.

The token IDs will then traverse all the transformer encoders. Each encoder is made up of two layers: a Self-attention layer and a feed forward neural network. Each encoder's inputs are fed to the self-attention layer first to compute the contribution of each word in the sentence to all the other words in the sentence and hence take into account the context of each word. The Self-attention layer passes its results through a feed-forward neural network and then hands it off to the next encoder.

As explained earlier, the Self-attention layer allows the encoder to look at other word positions in the input sentence for better encoding. For example, in the following commit message: "refactor out the abstract class OverlayEndpoint and use it to share functionality between OverlayTransport and service connection", the model should understand what the word "it" refers to? The self attention will associate "it" with "OverlayEndpoint "(the abstract class).

For text classification tasks, we only take the output corresponding to the [CLS] token for each input commit and present it to a classifier model along with its commit category to train and test commit classifiers.

4. Methodology

One of the main objectives of this paper is to conduct a deep investigation of the impact of introducing source code changes in the process of automatic commit classification. We propose to use six different models: (1) DNN@BERT: a deep neural network (DNN) model that is trained on BERT's word representations (BWR) of commit messages, (2) DNN@BERT+Refact_cc: a DNN model that combines BWR of commit messages and source code changes obtained by using RefactoringMiner, (3) DNN@BERT+CD_cc: a DNN model trained on a dataset that includes BWR of commit messages and source code changes generated by the ChangeDistiller tool, (4) DNN@BERT+Fix_cc: a DNN model that combines BWR of commit messages and source code changes obtained from Fixminer, (5) DNN@BERT+All_cc: a DNN model that combines BWR of commit messages and all code changes collected by all previously used source code distiller tools and finally, (6) DNN@KW+CD_cc: a DNN model trained on a dataset that includes a pre-determined list of keywords extracted from commit messages and source code changes generated by the ChangeDistiller tool.

All models have been trained on the same set of commits (i.e. the same instances) and using the same classification technique, namely, a DNN. Fig. 3 illustrates the overall process that we split into 4 phases: (a) Commit messages collection and code changes generation, (b) BERT Representations of commit messages, (c) Dataset Generation, and (d) DNN model training, validation and testing.

Given a set of labeled commits, where each commit is described by a set of code change-based binary features and a set of BERT features, we adopt a Deep Neural Network (DNN) [30] as a base classifier. Apart from the DNN, other classification techniques could be also used. We provide in the following a detailed description of the main phases of our approach.

4.1. *Commit messages collection and code changes generation*

In this section, we describe our data collection process and how we have collected our commit messages and the different source code changes for each commit.

4.1.1. *Collection of commit messages*

The commit messages used in this work were obtained from three publicly available datasets produced by Levin et al. [2], Mauczka et al. [24] and AlOmar et al. [31]. Their datasets contain commits collected from different open source projects. The selected projects cover a wide range of software domains such as IDEs, programming languages, distributed databases and integration frameworks.

4.1.2. *Generation of source code changes*

In this paper, we aim at analyzing and understanding how and when source code changes support the commit classification into different maintenance categories. To achieve this goal, we had to get a variety of fine-grained source code change types. To do so, we conducted a thorough study of existing tools which can automatically detect these code changes.

Several automatic tools have been proposed to understand how the software evolves and produces structural code change types. Among these tools, we mention: ChangeDistiller [14], a tool for extracting fine-grained source code changes that happened between subsequent revision of Java classes. Gumtree [13] is another tool inspired by the way developers manually search for changes between two file versions. Coming [15] is a generic tool that takes a git repository as input and

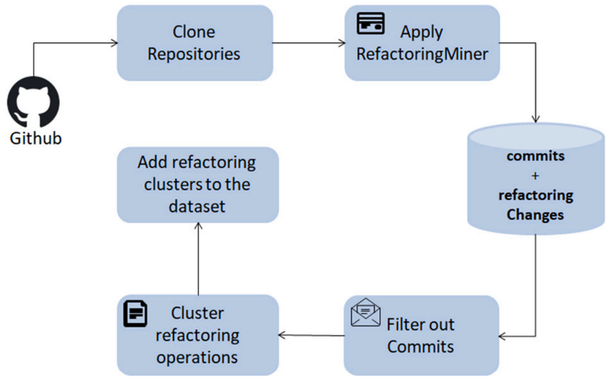
Clone Repositories
Apply RefactoringMiner
Github
commits + refactoring Changes
Add refactoring clusters to the dataset
Cluster refactoring operations
Filter out Commits

Fig. 4. Process of collecting refactoring changes.

mines instances of change patterns on each commit. Apart from that, there are other tools capable of detecting code refactoring produced by developers such as Ref-Finder [32] and Ref-diff [11].

The dataset used in Levin et al. [2] contains a set of code changes generated by ChangeDistiller tool. Hence, we also used ChangeDistiller to extract the same code changes from the commit messages that we collected from the other sources. We used two other tools to collect more code changes, namely, RefactoringMiner [12] and FixMiner [16].

Refactoring code changes

To extract refactoring code changes, we used RefactoringMiner [12], an automatic tool used to detect refactoring operations in the entire history of git repositories between specified commits. Our choice is motivated by the fact that it outperforms the other refactoring tools. It can easily detect more than 35 fine-grained change types and gives as output a huge number of commits and refactoring changes.

The process of using RefactoringMiner is illustrated in Fig. 4. First, we cloned the list of projects we have selected for this work from GitHub. Then we applied RefactoringMiner on each project repository. At this step, the tool starts collecting all the possible refactoring changes that have been made by developers during the development process.

After removing duplicate code changes, i.e. those that have been already generated by ChangeDistiller, we ended up with 23 types of code refactoring (listed in Table 3). We used the selected refactoring types as binary features (1 if the refactoring type happened in the commit, 0 otherwise).

Bug fix code changes

FixMiner [16], an automated tool for mining semantically relevant fix patterns from commits that refer to corrective changes. We used this tool to collect changes related to bug fixes.

The overall process of extracting bug fix patterns is illustrated in Fig. 5. First, we presented to FixMiner a csv file containing the list of projects to use. At this step, the mining tool first clones all the project repositories listed in the input file, then starts inspecting the necessary files and documents to distill fix patterns (i.e., commits and bug reports), then moves to the next step which consists in distilling all fix patterns that can be found in these documents. Finally we encoded the obtained patterns in the same way: if a fix pattern exists in a commit message then the "bug fix" feature will be assigned a value equal to 1, 0 otherwise.

ChangeDistiller code changes

We have also included a set of code changes used in the publicly available dataset of Levin et al. [2]. These changes are distilled as per the taxonomy defined by Fluri et al. [33]. It consists of 46 different change types (listed in Table 3). To obtain these source code changes, we re-used ChangeDistiller then checked if each one of these source

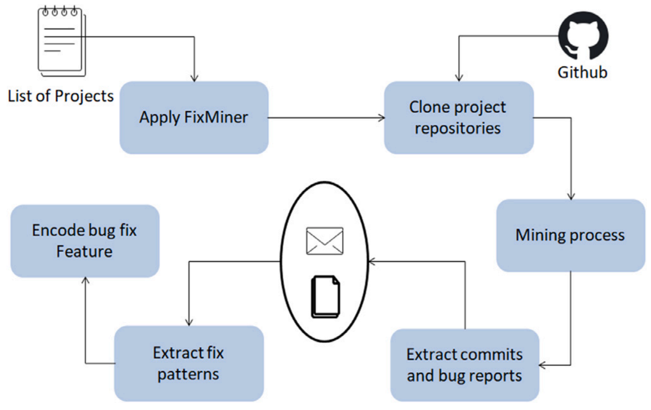

Fig. 5. Process of extracting bug fix patterns.

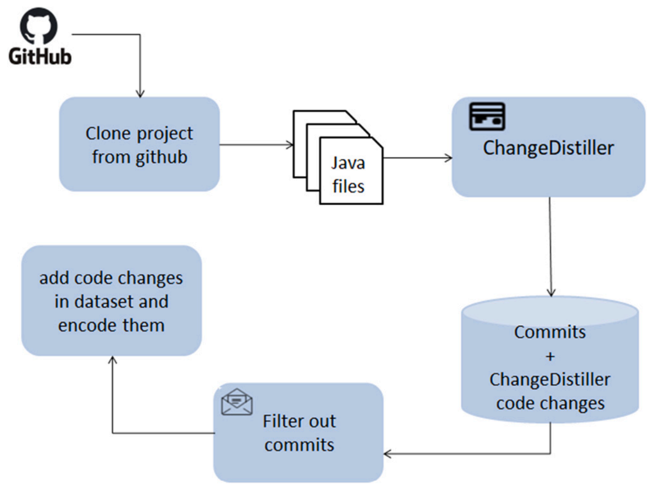
GitHub
Clone project from github
Java files
ChangeDistiller
add code changes in dataset and encode them
Commits + ChangeDistiller code changes
Filter out commits

Fig. 6. Process of collecting ChangeDistiller code changes.

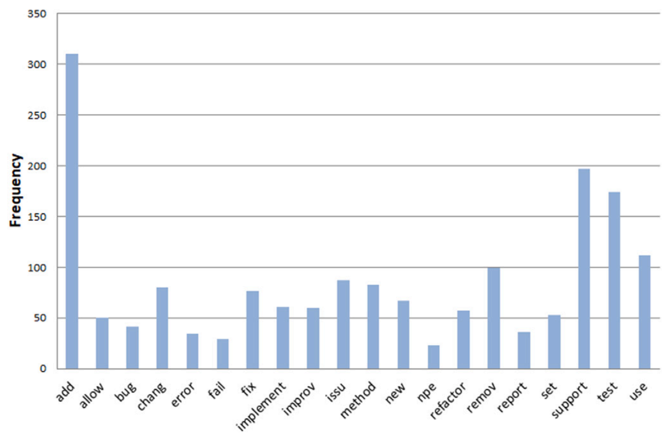

Fig. 7. Frequency of keywords in the entire dataset.

code changes happened or not in each commit. Fig. 6 describes the process of collecting ChangeDistiller code changes.

Keywords

Levin et al. [2] used a set of 20 keywords that they extracted from commit messages. Fig. 7 shows the list of the keywords as well as their frequencies in our entire dataset. We can see from the figure that the distribution of the keywords is not uniform. In fact, few keywords are more used than others by commit messages (e.g. add,

Table 3
List of adopted code change types with their frequencies in the entire dataset (1793 commits).

ChangeDistiller code changes
Adding Attribute Modifiability(7)/Method Overidability(6)
Additional Class(237)/Functionality(237)/Object State(751)
Alternative Part Delete(145)/Insert(238)
Attribute(82)/Method(148)/Class Renaming(4)
Attribute Type Change (85)
Comment Delete(218)/Insert(294)/Move(109)/Update(48)
Decreasing(103)/Increasing(158) Accessibility Change
Doc Delete(62)/Insert(177)/Update(204)
Parameter Delete(130)/Insert(218)/Ordering Change(16)/Renaming(67)/Type Change(106)
Parent Class Change(45)/Delete(9)/Insert(23)
Parent Interface Change(1)/Insert(45)/Delete(30)
Removed Functionality(347)/Class(255)/Object State(255)
Removing Attribute Modifiability(19)/Class Derivability(5)/ Method Overridability(9)
Return Type Change(141)/Delete(81)/Insert(11)
Condition Expression Change(388)
Statement Delete(730)/Insert(921)/Ordering Change(199)/ Parent Change(458)/Update(845)
RefactoringMiner code changes
Extract Method(153)/Class(24)/Subclass(5)/Superclass(33)/Interface(10)/variable(87)
Extract and move method(40)
Inline Method(23)/Variable(9)
Rename Variable(116)
Move Method(62)/Attribute(39)/Class(5)/Source folder(7)
Pull Up Method(33)/Attribute(25)
Push Down Method(8)/Attribute(6)
Change Package(12)
Parametrize Variable(17)
Move and rename Class(18)
Replace Variable with attribute(16)
Change variable type(66)
Fixminer code changes
Code patches(123)

Table 4
Annotation guide design.

Commit content	Label category
Refactoring changes (e.g., renaming, moving, pulling up, etc.) and code clean up	Perfective
Security, docs and comments related commits	Perfective
Performance tests (e.g., regression and unit tests, etc.)	Perfective
Time execution and memory usage optimizations	Perfective
Adding new functionalities, modules and methods	Adaptive

Table 5
Dataset characteristics.

Feature	#Features
Refactoring code change features	23
Bug fix code change features	1
ChangeDistiller code change features	46
Keyword features	20
Total	**90**
Class label	#Instances
Corrective	600
Adaptive	590
Perfective	603
Total	**1793**

support, test, use). Since our dataset is balanced, this might explain that these keywords exist in commits that do not necessarily belong to the same category. We have reused these keywords as binary features (1 if the keyword exists in the commit message, 0 otherwise) by the DNN@KW+CD_cc model for comparison purposes.

4.1.3. Commits labeling

In this work, we are using a multi-class classification approach where each commit is assigned a single label (commit category) from the set: {corrective, adaptive and perfective}.

Commit messages that we have collected from the dataset used by Levin et al. [2] are already labeled by these categories. We have also taken commits out from Mauczka et al.'s dataset [24] which contains multi-labeled commits using the same three categories. We have only selected singly-labeled commits from this dataset. As for the commits collected from AlOmar et al.'s dataset [31] which is not labeled, we have selected commit messages (belonging to only adaptive and perfective categories) that all authors agreed on their labels. To help annotate these commits, the authors followed some rules of thumb to assign the labels to the different commits (see Table 4).

The data collection phase resulted in a dataset of 1793 labeled commit messages. In addition to the commit messages, we also have a set of 70 code change-based features as well as 20 keyword-based features extracted from the commit messages. A summary of the characteristics of the used dataset is provided in Table 5.

4.2. BERT's words representations of commit messages

As shown in Fig. 3(b), all commit messages will be fed to the pretrained BERT model to produce the BERT's word representations (BWR) of each token in the message. As detailed in Section 3.3, the output corresponding to the [CLS] token for all commit messages will result in a new set of 768 features.

4.3. Dataset generation

This step consists of preparing the final dataset that will be processed by the DNN classifier. As shown in Fig. 3(c), the dataset contains a set of 768 BERT-generated features, a set of 70 code change-based features and a set of 20 keyword-based features. The list of features to use differs from one model to another. For instance, only ChangeDistiller code changes and BERT's features are used to train the DNN@BERT+CD_cc model.

Table 6
Optimal hyper-parameters values for the different used DNN models.

Model	#Hidden layers	#Hidden units	#Epochs	Batch size	Dropout	Learning rate	Val-accuracy
DNN@BERT	2	300	100	32	0.2	0.001	78%
DNN@BERT+Fix_cc	2	400	100	32	0.2	0.001	78.89%
DNN@BERT+Refact_cc	4	500	100	32	0.2	0.001	76.11%
DNN@BERT+CD_cc	3	400	100	32	0.2	0.001	77.78%
DNN@BERT+All_cc	3	300	100	32	0.2	0.001	73.33%
DNN@KW+CD_cc	2	76	100	32	0.2	0.001	73.33%

4.4. DNN training, validation and testing

This step consists in training a deep neural network (DNN) model that will be used to classify the future unseen commits given their corresponding vector of feature values.

We implemented 6 fully connected DNNs using Keras (version 2.2.5), one for each of the 6 models (i.e., DNN@BERT+{Refact_cc, CD_cc, Fix_cc, All_cc, KW+CD_cc}).

In order to build the DNNs, we adopted the sequential model and we used the Dense library. The input layer of the DNN@KW+CD_cc model consists of 66 input neurons (as we have 20 keyword-based features and 46 change types obtained from ChangeDistiller). For the DNN@BERT model, we have 768 input neurons (as we have 768 BERT-based features). In the other hand, we proceed with 769 input neurons in DNN@BERT+Fix_cc model, 838 input neurons in DNN@BERT+All_cc, 814 input neurons in DNN@BERT+CD_cc and finally, 791 input neurons in DNN@BERT+Refact_cc model.

In order to select the best architecture for each one of these models, we have tested different numbers of hidden layers (2 to 4). In each hidden layer, we used the same number of neurons. For DNN@KW+CD_cc, we tested different numbers of hidden units going from 36 to 96. As for the remaining models using BERT, we tested different numbers of hidden units going from 200 to 700. The output of each hidden neuron is calculated as the weighted sum of its inputs. The summation is then passed to a Rectified Linear Unit (ReLU). For the output layer, the Softmax activation function is used.

For all the models, Adam [34] was used as the learning rate optimization algorithm. We tested different values of learning rate {0.001, 0.002, 0.003, 0.004, 0.005}. We also tested batch-size values of 32 and 64. In order to reduce overfitting, L2 regularization is used and the dropout parameter was varied in {0.1, 0.2, 0.3, 0.4, 0.5}. For the 6 models, we opted for 100 epochs.

As for the testing strategy, we split the dataset into 70% for training, 10% for validation and 20% for testing. For each model, the best combination of the aforementioned hyper-parameters is obtained by training it on the training set and evaluating it on the validation set.

Based on the best validation performance, we define our model with the best hyper-parameter combination then re-train it on 80% of the dataset (training + validation sets), and test it on the initially selected testing set.

In order to use the DNN@BERT model to classify a new commit, the corresponding commit message is fed to BERT in order to produce its BERT based-features then this new commit representation will be fed to the DNN classifier to get its category.

When considering a new commit message, all the models, except the basic one (i.e., DNN@BERT) need to extract the corresponding source code changes using the appropriate change mining tools and then integrate them in the model classification.

However, for the DNN@KW+CD model, each new commit is checked against the list of pre-determined keywords and code change types (as explained above) then fed back to the classification model.

5. Experimental results and analysis

In this section, we describe the experimental setup then we report and analyze results of several experiments that investigate the efficacy of the aforementioned models on the commit classification performance.

Table 7
The different versions of the dataset.

Dataset	Features
Dataset_{BERT}	BERT's words representations (BWR)
$\text{Dataset}_{BERT+Fix\_cc}$	BWR + Fixminer code change
$\text{Dataset}_{BERT+Refact\_cc}$	BWR + RefactoringMiner code changes
$\text{Dataset}_{BERT+CD\_cc}$	BWR + ChangeDistiller code changes
$\text{Dataset}_{BERT+All\_cc}$	BWR + Fixminer + RefactoringMiner + ChangeDistiller code changes

5.1. DNN models' hyper-parameters

We started our experiments with a hyper-parameters tuning step (as described in Section 4.4) for the different DNN models that we are using in this work. Each model has been trained on the training set then evaluated on the validation set. In Table 6, we provide the hyper-parameters values (best combination of values) for the different selected models based on the highest obtained validation accuracy. In the subsequent experiments, the DNN configurations provided in Table 6 are used.

5.2. Results and analysis

In this section, we present experimental results and analyses with the aim to answer the following research questions:

RQ1: How do code changes help improve commit classification when used with the pre-trained BERT model?

In RQ1, our aim is to evaluate the impact of source code changes in commit classification when we combine them with the generated BERT's words representations (BWR) features, and check how code changes extracted by different change mining tools impact each of the 3 commit categories differently. We will also determine the best performing commit classification model. To answer this question, we generated 5 dataset versions from our original dataset. In each dataset version, we kept a subset of features as shown in Table 7.

RQ2: How does the best performing BERT-based model compare to the state-of-the-art combined (keywords −code changes) model?

In RQ2, our purpose is to compare the best performing model, namely, DNN@BERT+Fix_cc to the combined model proposed in [2]. To achieve this, we trained, fine-tuned and tested the DNN@KW+CD_cc model as well as other standard classifiers using $\text{Dataset}_{KW+CD\_cc}$. In this research question we will also see how standard classifiers perform on the $\text{Dataset}_{BERT+Fix\_cc}$ when compared to DNN-based models.

5.2.1. RQ1: How do code changes help improve commit classification when used with the pre-trained BERT model?

This research question focuses on checking whether code changes, when combined with BWR features that we derive from pre-trained BERT model, can achieve a good commit classification performance.

For this experiment, we adopted different performance measures, namely, accuracy (percentage of correct classifications) and per-class precision, recall and f1-scores. Since our problem is a multi-class classification problem and that our dataset does not suffer from class

Table 8
Accuracy and Macro-average f1-score of DNN-based models.

Model	Accuracy	F1-score
DNN@BERT	77.15%	0.77
DNN@BERT+Fix_cc	**79.66%**	**0.80**
DNN@BERT+Refact_cc	74.65%	0.75
DNN@BERT+CD_cc	72.15%	0.72
DNN@BERT+All_cc	70.2%	0.70

imbalance, we will also report the macro-average f1 score Avg_f1_score which represents the mean of the f1-scores of the n class labels:

$$Avg\_f1\_score = \frac{\sum_{i=1}^{n} f1 - score(L_i)}{n} \quad (1)$$

Table 8 shows the accuracy and the average f1-score values of DNN based-models. DNN@BERT model reaches an accuracy of 77.15% and an average f1-score equal to 0.77. This performance proves that the large vocabulary used by BERT can be useful to capture context in commit messages. The best performing model, DNN@BERT+Fix_cc, achieved an accuracy of 79.66% and f1-score up to 0.80.

When positioning DNN@BERT+Fix_cc against DNN@BERT, we find out that using Fixminer code change in addition to BWR (BERT) may be helpful to improve the classification of corrective commit category. As shown in Table 9, DNN@BERT+Fix_cc achieves a precision of 0.77 and f1-score of 0.81 for corrective class, which is better than respectively the 0.71 and 0.77 obtained by DNN@BERT for the same class. A concrete example is shown in the following commit message:

“IDEA-78863 Live templates are not accessible in- read-only file (e.g. versioned in Perforce or TFS)”.

This commit describes and fixes an error related to the non-accessibility of templates, when files are read-only. While the message implicitly conveys the inability of a functionality to be correctly executed, DNN@BERT misclassifies it. However, FixMiner has flagged the application of a corrective patch, therefore, when the text classification was augmented with the feature of detecting fixes, the DNN@BERT+Fix_cc has correctly classified the commit into corrective. Similarly, in many commit messages, BERT’s vocabulary was found to be not sufficient to return the correct label, especially when commits do not contain typical tags, such as fix, and bug. The augmented DNN@BERT+Fix_cc was not only able to outperform DNN@BERT, but also, it outperforms all the other augmented models, as shown in Table 9. According to Fig. 8, DNN@BERT+Fix_cc has only misclassified 18 bug fix commit instances, achieving the lowest false positive rate across all models. Class labels are named in Fig. 8 as follows: 0: adaptive, 1: corrective and 2: perfective.

Similarly to Levin et al. [2], we challenge the ability of augmenting a model with distilled code changes, to improve its accuracy. According to Table 9, DNN@BERT+CD_cc gives an accuracy of 72.15% and f1-score of 0.72. Interestingly, DNN@BERT outperforms its augmented DNN@BERT+CD_cc. However, while the overall performance of DNN@BERT is better than DNN@BERT+CD_cc in all classes aggregated, the use of code changes in addition to BERT’s BWR, has improved its performance in classifying adaptive commits, the only class in which DNN@BERT+CD_cc has performed better than DNN@BERT. This augmentation allows the classifier to reach a recall of 0.84 for the adaptive class, which explains the increased number of correctly classified adaptive instances (88 for DNN@BERT vs. 99 for DNN@BERT+CD_cc). Taking a look at the misclassified examples, we can showcase the following:

“refactor sub fetch phase to also allow for hits- level execution”.

This commit is correctly classified, by DNN@BERT+CD_cc, as adaptive, and misclassified, by DNN@BERT, as perfective. The commit message contains the keyword refactor, which is a strong feature that

Table 9
Per-class precision, recall and f1-score of DNN-based models.

Class	Precision	Recall	F1-score
DNN@BERT			
Adaptive	0.83	0.75	**0.79**
Corrective	0.71	0.84	0.77
Perfective	0.79	0.72	0.76
DNN@BERT+Fix_cc			
Adaptive	0.84	0.75	**0.79**
Corrective	0.77	0.85	**0.81**
Perfective	0.79	0.79	**0.79**
DNN@BERT+Refact_cc			
Adaptive	0.76	0.73	0.74
Corrective	0.76	0.79	0.77
Perfective	0.72	0.72	0.72
DNN@BERT+CD_cc			
Adaptive	0.66	0.84	0.74
Corrective	0.77	0.66	0.71
Perfective	0.76	0.67	0.71
DNN@BERT+All_cc			
Adaptive	0.75	0.64	0.69
Corrective	0.69	0.75	0.72
Perfective	0.67	0.71	0.69
DNN@KW+CD_cc			
Adaptive	0.60	0.71	0.65
Corrective	0.71	0.58	0.64
Perfective	0.66	0.67	0.66

characterizes the perfective class messages. However, in this message, the keyword is being misused, as the developer is adjusting an existing feature. This is refelected at the source code level, as there is only one rename refactoring operation, while several behavior changing actions were taken, including *Additional functionality*, *additional object state*, *statement update/insert/delete*, *comment move/insert*, *statement parent change*, *statement ordering change*, *return type insert*, *return type change*, *parameter type change*, *parameter ordering change*, *parameter insert*, etc. These code changes clearly support the adaptive context of this commit, which subsequently improves the classification quality of adaptive category. This example showcases the limitation of purely relying on textual descriptions, to classify code changes. Textual descriptions can not only be incomplete and ambiguous, they can also be misleading, like the case of the refactor keyword misuse, that has been also confirmed in prior studies [35,36], which hardens the reliance on keywords alone.

Since refactorings represent the intuitive characterization of perfective changes, it is expected that DNN@BERT+Refact_cc would achieves a good accuracy in classifying perfective commits. Surprisingly, DNN@BERT+Refact_cc overall accuracy was 74.65% and f1-score was 0.75, placing it third behind DNN@BERT and DNN@BERT+Fix_cc. These latter models have also outperformed DNN@BERT+Refact_cc in the perfective class, which was expected to be accurately predicted by this model. Looking at the misled testing classified instances, we notice that using RefactoringMiner code changes increases the confusion of the model, in all classes, and not only the perfective class. The analysis of refactoring frequencies, show a uniform distribution of these operations, across commits, belonging to all classes. Therefore, the discriminative nature of these code changes, as features, are weak. To concretely understand this situation, we analyzed a few misclassified examples. In the following example:

“Features: Added http file downloader, integrated into TVRageProvider - Implemented ‘check for updates’ and preference option Code changes: Finally made preferences static (yay!) - Renaming of many UIStarter variables - Moved string handling into StringUtils - Updated tests”.

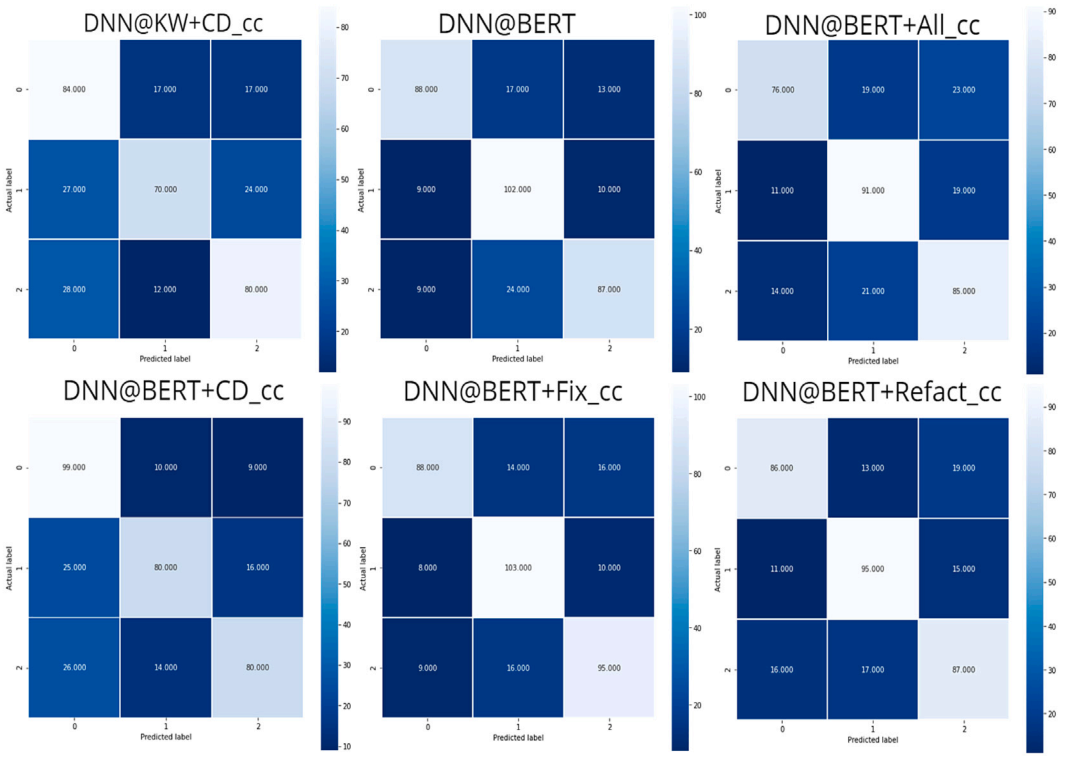

Fig. 8. Confusion matrices of DNN based models.

As explained in the commit text, the developer has interleaved refactoring practices with other development-related tasks, i.e., adding feature. Specifically, the developer implemented two new functionalities (i.e., allow the user to download files, and "check for updates" and "preference option" features. Developers also performed other code changes which involved renaming, moving, etc.

The analysis of this commit's code changes portrays the 23 refactoring operations performed in which the developer added features and made other related code changes. With regards to the type of refactoring operations used to perform these implementations, the developer mainly performed moving and renaming related operations that are associated with code elements related to that implementation. Overall, *Rename Variable* and *Rename Attribute* constitute the main refactoring operations performed accounting for 38.10% and 28.57% respectively, followed by *Move Class* with 14.29% and *Rename Method* with 9.52%. The percentage of *Move Method* and *Move and Rename Class* refactorings, by contrast, made up a mere 4.76%.

Upon exploring the source code, it appears to us that the developer performed moving-related refactorings when adding features (e.g., update checker functionality and activate user preference option) to the system, and renaming-related operations have been performed for several enhancements related to the UI (e.g., renaming buttons, task bar, progress bar, etc.). These observations may explain that adding feature is one type of development task that refactorings were interleaved with and the refactoring definition in practice seems to deviate from the rigorous academic definition of refactoring, i.e., refactoring to improve the design of the code.

If we consider this example:

"Several UI-related bugs solved. Also corrected the storing/retrieving…".

This case study presents another refactoring intention, i.e., refactoring to fix bugs that differs from the academic definition of refactoring. It can be seen from the above commit message that several UI-related bugs have been solved while performing refactorings. Similar to the previous example, the developer interleaved these changes with other types of refactoring.

In this commit, 7 distinct refactoring operations performed that constituted 37 refactoring instances for bug fixing-related process. The type of refactorings involved in this activity are mainly focused on extracting, moving, and renaming-related operations. Roughly a quarter of refactorings were *Move Method. Rename Parameter, Rename Attribute, and Move Attribute* constituted almost the same percentage with slight advantage to *Rename Parameter*. *Extract Method* was comprised of 13,51%, whereas *Rename Variable, Rename Method*, and *Extract Class* combined just constituted under a fifth. With the detected amount of refactoring, for a corrective commit, DNN@BERT+Refact_cc was heavily influenced and ended up misclassifying the commit. We conclude from the previous examples that developers flossly refactor the code to reach a specific goal, i.e., fix bugs, adding features, and not only when improving the design (perfective class). Interleaving these activities weakens the discriminative nature of this type of code change, as a feature that can be exclusive to a given class, i.e., perfective.

Finally, DNN@BERT+All_cc, whose average f1-score is 0.70 shows inferior performance in comparison with DNN@BERT's f1-score. The rationale behind augmenting BERT with all possible code change distillers and pattern miners, is driven by the assumption of some change patterns can be learned by the classifier, to help distinguish between the classes. However, we can clearly see that this is not practically the case. Interestingly, when using the results of ChangeDistiller, FixMiner, and RefactoringMiner, DNN@BERT+All_cc performs the worst across all models. For instance, as discussed above, refactorings can exist in multiple classes and not only in the perfective class as we see in the following example:

"Implemented paymentview tabs using BaseTabHandler, added dialog for the quick dial codes and the refactored code".

This commit is actually adaptive and misclassified into perfective because it contains some refactoring code changes: *Extract Class*, *Move Method* and *Move Attribute*. Moreover the commit description of this example can confuse the model even it uses BWR, since there are different words that indicate both adaptive (i.e., "Implemented", "added") and perfective (i.e., "refactored") categories.

Table 10
Optimal hyper-parameters of standard classifiers trained on Dataset$_{BERT+Fix\_cc}$.

Classifier	Hyper-parameters
RF	n_estimators = 200, max_features = 40, max_depth = 110, min-samples_split = 10
SVM	c = 100, gamma = 0.0001, kernel = 'rbf'
KNN	n_neighbors = 11, weights = distance, metric = 'euclidian'
GBM	learning_rate = 0.15, n_estimators = 100, max_features = 'sqrt', max_depth = 90
DT	max_depth = 100, max_features = 60, criterion = 'gini'

Table 11
Accuracy of DNN@BERT+Fix_cc vs. other classifiers on Dataset$_{KW+CD\_cc}$.

Model	Accuracy
DNN@BERT+Fix_cc	**79.66%**
DNN@KW+CD_cc	65.18%
NB@KW+CD_cc	55%
SVM@KW+CD_cc	65.46%
GBM@KW+CD_cc	65.74%
RF@KW+CD_cc	71.98%
KNN@KW+CD_cc	65%
DT@KW+CD_cc	64.91%

5.2.2. RQ2: How does the best performing BERT-based model compare to the state-of-the-art combined (keywords+code changes) model?

To answer this research question, we will compare results of the DNN@BERT+Fix_cc with results of DNN@KW+CD_cc: a DNN model trained on commits described by the combination of keywords and code changes used in [2]. In addition to the deep neural networks, we also adopted standard classification techniques used in [2], namely, K-nearest neighbors (KNN), C4.5 Decision Tree algorithm (DT), Random Forests (RF) and Gradient Boosting Machines (GBM). We also considered Support Vector Machines (SVM) and a probabilistic method (Naive Bayes (NB)). Experiments with these classifiers have been carried out using the Scikit-learn python library [37]. Like with the DNN models, we performed a fine-tuning step (using the same validation strategy described in Section 4.4) of the hyper-parameters of each one of the aforementioned standard classifiers. The Scikit-learn's GridSearch() function has been used for this fine-tuning step. Table 10 provides the hyper-parameters values corresponding to the best performing model. For the KNN classifier, we have tested several distances for real-valued vectors, namely, Euclidean, Manhattan, Chebyshev and Weighted Jaccard. Best accuracy with KNN has been obtained with the Euclidean distance (65%) followed by Manhattan distance (62%) then Chebyshev (56.83%) then Weighted Jaccard (54.42%).

Accuracy results in Table 11 show that DNN@BERT+Fix_cc outperforms models that use keywords combined with code changes. When looking at the corrective category in Table 9, we notice that DNN@BERT+Fix_cc is performing better than DNN@KW+CD_cc in precision (0.77 vs. 0.71), recall (0.85 vs. 0.58) and f1-score (0.81 vs. 0.64). This can be explained by the fact that commit descriptions which are written by developers are complex by nature. This makes their analysis more challenging, especially for corrective commits which consistently contain what FixMiner detects as patches. This may support the classification when the message does not contain typical keywords that discriminate for the corrective class. Here is an example of a commit, labeled by its author as corrective:

"HBASE-5857 RIT map in RS not getting cleared- while region opening"

DNN@KW+CD_cc model was not able to make the best decision since it completely relies on commit description. On the other hand, FixMiner code change intervenes with BWR to eliminate ambiguities and help the model correctly classify instances belonging to the corrective category.

Table 12
Accuracy of DNN vs. standard classifiers on Dataset$_{BERT+Fix\_cc}$.

DNN	SVM	NB	GBM	RF	KNN	DT
79.66%	61.83%	73%	67.13%	72.18%	62.12%	60%

To justify the selection of the DNN model as a base classifier for our approach, we show in the next experiment results of the DNN model and the previously used standard classifiers on Dataset$_{BERT+Fix\_cc}$. Results in Table 12 show that the DNN-based model outperforms all other classifiers.

6. Threats to validity

This section enumerates all the factors that may influence the correctness of our findings.

Our analysis is mainly threatened by the accuracy of the tools we used to extract the code changes. If these tools miss any code changes, or report inaccurate ones, this will negatively impact our results. However, we selected these tools as they are known for their accuracy, for instance, refactoring studies [12,38] report that Refactoring Miner has high precision and recall scores in comparison with off-the-shelf refactoring detectors. Similarly, FixMiner's probability of detecting fix patterns is up to 80%. Also, as we were using these tools, we did not notice, in the examples we manually examined, any anomaly or inconsistency in the mined changes. Also, our approach heavily relies on the quality of commit messages. To reduce the impact of quality and quantity of commit messages, and to perform a more accurate comparison, we reused the same set of well-commented Java projects, experimented in previous studies.

Another main threat relates to the sample bias, as the size and type of data may not be representative. To mitigate this risk, we have chosen commits from various projects, to diversify the sources. Moreover, the dataset used in this paper is perfectly balanced, i.e., the selected commits are evenly distributed between the three maintenance categories, which does not mimic real world settings, where these developer contributions will not be evenly distributed between these classes. We have opted for giving all classes the same number of commits, because we do not want the difference in class distribution to interfere with our comparison of the various classifiers. Yet, using perfectly balanced dataset (i.e. each category contains exactly the same number of instances) both for training and testing may not be applicable in real-world scenarios, where the test dataset should represent a more realistic distribution of the labels. Table 13 reports the distribution of labels, per class, in state-of-the-art studies.

In Table 13, we notice that the distribution of classes is different from one dataset to another and that the perfective class is not always the dominant class in all studies. Keeping in mind that our best performing model (DNN@BERT+Fix_cc) has scored its worst F1-score for this class, then using less perfective testing instances would eventually decrease the probability of falsely labeling a perfective commits, and so, it would potentially increase the overall performance of our model. So, using a balanced setting was not necessarily in favor of our model.

During the labeling process, we considered only commits where a consensus between authors was made about whether a message is clearly conveying towards one class. Unclear and debatable commits were discarded. Our dataset is also available online for extension and refinement.

Relying on only open-source projects limits the generalization of our findings, since corporate companies typically enforces the use of standards when annotating code changes. Similarly, using popular repositories may not be representative to all open-source systems.

Table 13
Distribution of labels, per class, in previous studies.

Study	Corrective	Adaptive	Perfective
Hindle et al. [4,22]	9%	33%	32%
Mauczka et al. [24]	31%	40%	26%
Yan et al. [26]	33%	33%	33%
Levin and Yahudai [2]	43%	21%	35%
Zafar et al. [21]	100%	N/A	N/A

7. Conclusion and future work

In this paper, we presented and compared several models for commit classification. Our approach combines BERT's word representations with the fine-grained code changes extracted from the source code for each commit. To do so, we used existing code change mining tools to identify fine-grained code changes, refactoring operations, and bug patches. We trained, validated and tested our deep neural network based models on a labeled dataset that we created from a variety of open source projects. Our key findings show the effectiveness of our model (DNN@BERT+Fix_cc) in correctly predicting the type of the commit change, scoring an accuracy of almost 80% and an f1-score of 0.8 despite the relatively small size of the dataset. We believe this initiative shows a significant advance in classifying commits, and which can be easily extended by the community if more commits or a different set of relevant features are considered.

In the future, we plan on including more labeled commits from other projects. We believe that the performance of the BERT+DNN models will increase when a larger dataset is used. We also plan on designing a language-agnostic model that operates for multiple languages, including Python, and C#. Our current study is limited to Java projects since it is the only language supported by the available code distillers [3]. Furthermore, we plan to deploy more software artifacts, such as change logs, issues, bug reports, as other dimensions to support the analysis of commit messages. In addition, we plan on using software differencing techniques [13,39] to extend the granularity of changes that occur across commits.

CRediT authorship contribution statement

Lobna Ghadhab: Methodology, Software, Investigation, Data curation, Visualization, Writing - original draft. **Ilyes Jenhani:** Validation, Investigation, Formal analysis, Writing - original draft, Supervision. **Mohamed Wiem Mkaouer:** Conceptualization, Resources, Formal analysis, Validation, Writing - original draft. **Montassar Ben Messaoud:** Project administration, Formal analysis, Writing - original draft, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] R.V.R. Mariano, G.E. dos Santos, M.V. de Almeida, W.C.B. ao, Feature changes in source code for commit classification into maintenance activities, in: 18th IEEE International Conference On Machine Learning And Applications, ICMLA, 2019, pp. 515–518.

[2] S. Levin, A. Yehudai, Boosting automatic commit classification into maintenance activities by utilizing source code changes, in: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, in: PROMISE, ACM, New York, NY, USA, 2017, pp. 97–106.

[3] S. Gharbi, M.W. Mkaouer, I. Jenhani, M. Ben Messaoud, On the classification of software change messages using multi-label active learning, in: Proceedings of the 34rd Annual ACM Symposium on Applied Computing (SAC), 2019, pp. 1760–1767.

[4] A. Hindle, D.M. German, M.W. Godfrey, R.C. Holt, Automatic classification of large changes into maintenance categories, in: IEEE 17th International Conference on Program Comprehension, 2009, pp. 30–39, http://dx.doi.org/10.1109/ICPC.2009.5090025.

[5] S. Chakraborty, M. Allamanis, B. Ray, Tree2tree neural translation model for learning source code changes, 2018, CoRR.

[6] K. Herzig, S. Just, A. Rau, A. Zeller, Predicting defects using change genealogies, in: 24th International Symposium on Software Reliability Engineering (ISSRE), 2013, pp. 118–127.

[7] E.G. Knyazev, Automated source code changes classification for effective code review and analysis, in: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, 2008.

[8] P. Weissgerber, S. Diehl, Identifying refactorings from source-code changes, in: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), IEEE, 2006, pp. 231–240.

[9] Y. Zhou, A. Sharma, Automated identification of security issues from commit messages and bug reports, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, 2017, pp. 914–919.

[10] A. Mockus, L.G. Votta, Identifying reasons for software changes using historic databases, in: International Conference on Software Maintenance (ICSM), 2000, pp. 120–130.

[11] D. Silva, M.T. Valente, Refdiff: detecting refactorings in version histories, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 269–279.

[12] N. Tsantalis, M. Mansouri, L.M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (Eds.), ICSE, ACM, 2018, pp. 483–494.

[13] J. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 2014, pp. 313–324.

[14] B. Fluri, M. Würsch, H.C. Gall, Change distilling: Tree differencing for fine-grained source code change extraction, IEEE Trans. Softw. Eng. (2007) 725–743.

[15] M. Martinez, M. Monperrus, Coming: a tool for mining change pattern instances from git commits, in: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, 2019, pp. 79–82.

[16] A. Koyuncu, K. Liu, T.F. Bissyandé, D. Kim, J. Klein, M. Monperrus, Y.L. Traon, Fixminer: Mining relevant fix patterns for automated program repair, 2018, CoRR.

[17] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, in: NAACL-HLT (1), 2019.

[18] E.B. Swanson, The dimensions of maintenance, in: Proceedings of the 2nd International Conference on Software Engineering, ICSE, 1976, pp. 492–497.

[19] S. Levin, A. Yehudai, Using temporal and semantic developer-level information to predict maintenance activity profiles., in: IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, IEEE Computer Society, 2016, pp. 463–467.

[20] J.J. Amor, G. Robles, J.M. Gonzalez-Barahona, A.N. Gsyc, Discriminating development activities in versioning systems : A case study, in: PROMISE'06, 2006.

[21] S. Zafar, M.Z. Malik, G.S. Walia, Towards standardizing and improving classification of bug-fix commits, in: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, 2019, pp. 1–6.

[22] A. Hindle, D.M. German, R. Holt, What do large commits tell us?: A taxonomical study of large commits, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, in: MSR '08, ACM, New York, NY, USA, 2008, pp. 99–108.

[23] N. Tsantalis, V. Guana, E. Stroulia, A. Hindle, A multidimensional empirical study on refactoring activity, in: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, in: CASCON '13, IBM Corp., Riverton, NJ, USA, 2013, pp. 132–146.

[24] A. Mauczka, F. Brosch, C. Schanes, T. Grechenig, Dataset of developer-labeled commit messages, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 490–493, http://dx.doi.org/10.1109/MSR.2015.71.

[25] L.P. Hattori, M. Lanza, On the nature of commits, in: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, 2008, pp. 63–71, http://dx.doi.org/10.1109/ASEW.2008.4686322.

[26] M. Yan, Y. Fu, X. Zhang, D. Yang, L. Xu, J.D. Kymer, Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project, J. Syst. Softw. 113 (Supplement C) (2016) 296–308.

[27] P. Loyola, E. Marrese-Taylor, Y. Matsuo, A neural architecture for generating natural language descriptions from source code changes, in: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL), 2017, pp. 287–292.

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, 2017, pp. 5998–6008, CoRR.

[29] V. Sanh, L. Debut, J. Chaumond, T. Wolf, Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2019, CoRR.

[30] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2018.

[31] E. AlOmar, M.W. Mkaouer, A. Ouni, Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages, in: 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), IEEE, 2019, pp. 51–58.

[32] M. Kim, M. Gee, A. Loh, N. Rachatasumrit, Ref-Finder: a refactoring reconstruction tool based on logic query templates, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, pp. 371–372.

[33] B. Fluri, H.C. Gall, Classifying change types for qualifying change couplings, in: 14th International Conference on Program Comprehension (ICPC), 2006, pp. 35–45.

[34] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: 3rd International Conference on Learning Representations (ICLR), 2015.

[35] D. Zhang, L. Bing, L. Zengyang, P. Liang, A preliminary investigation of self-admitted refactorings in open source software (s), in: The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 165-168, 2018, 2018, pp. 165–168, http://dx.doi.org/10.18293/SEKE2018-081.

[36] E.A. AlOmar, M.W. Mkaouer, A. Ouni, Toward the automatic classification of self-affirmed refactoring, J. Syst. Softw. 171 (2020) 110821.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[38] D. Silva, N. Tsantalis, M.T. Valente, Why we refactor? Confessions of github contributors, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), ACM, New York, NY, USA, 2016, pp. 858–870.

[39] M.J. Decker, M.L. Collard, L.G. Volkert, J.I. Maletic, Srcdiff: A syntactic differencing approach to improve the understandability of deltas, J. Softw. Evol. Process (2019).