Neural Computing and Applications (2022) 34:3981–4001
https://doi.org/10.1007/s00521-021-06659-3

ORIGINAL ARTICLE

Just-in-time software defect prediction using deep temporal convolutional networks

Pasquale Ardimento[1] · Lerina Aversano[2] · Mario Luca Bernardi[2] · Marta Cimitile[3] · Martina Iammarino[2]

Received: 10 November 2020 / Accepted: 27 October 2021 / Published online: 14 November 2021

Abstract
Software maintenance and evolution can introduce defects in software systems. For this reason, there is a great interest to identify defect prediction and estimation techniques. Recent research proposes just-in-time techniques to predict defective changes just at the commit level allowing the developers to fix the defect when it is introduced. However, the performance of existing just-in-time defect prediction models still requires to be improved. This paper proposes a new approach based on a large feature set containing product and process software metrics extracted from commits of software projects along with their evolution. The approach also introduces a deep temporal convolutional networks variant based on hierarchical attention layers to perform the fault prediction. The proposed approach is evaluated on a large dataset, composed of data gathered from six Java open-source systems. The obtained results show the effectiveness of the proposed approach in timely predicting defect proneness of code components.

Keywords Software fault prediction · Software quality · Software fault · Deep learning

1 Introduction

Due to the necessity of always more complex systems, software maintenance and evolution are critical and continuous activities that likely introduce new defects [7]. Consequently, an increasing interest is pointed toward the analysis and prediction of the effort and cost required for bug fixing since it can reduce resource waste and support decision-making. Some techniques allow checking if new software changes introduced new defects in the source code. For example, in the last years, several studies propose test cases [51] and code reviews [1]. While other approaches are based on statistical models exploiting source code and development process data identifying the software components that are more prone to be defective [52]. Specifically, there are two main types of component defectiveness prediction techniques: long-term and just-in-time (JIT). The long-term prediction uses information accumulated along software releases to identify the artifacts that are more prone to be defective. For example, some authors [10] propose and evaluate Object-Oriented metrics [23] to predict post-release defects, while other authors propose process metrics [6, 33]. A drawback of long-term defect prediction models is that in real scenarios they are poorly useful since they do not timely support [41] the developers when they are committing defectiveness code into the repository. Differently, JIT techniques exploit the characteristics of a code-change and eventually mark it as a fix-inducing change. This allows developers to deal with the defects as soon as they are introduced, indeed, recover design decisions later may be more and more difficult and the inspection activities require more effort. Overall, JIT approaches reduce this effort allowing early identification of the problems. With these approaches, the developers can reduce the time required to diagnose problems since they focus on the committed artifacts only [41].

✉ Pasquale Ardimento
pasquale.ardimento@uniba.it

[1] Department of Computer Science, University of Bari Aldo Moro, Bari, Italy

[2] Department of Engineering, University of Sannio, Benevento, Italy

[3] Department of Law and Economics, Unitelma Sapienza of Rome, Rome, Italy

This study, differently from the existing JIT defect prediction models [22, 35, 41, 52, 69, 70, 72], introduces a variant of the temporal convolutional networks (TCN), that exploits hierarchical attention layers, to predict which code changes are fix-inducing (i.e., are more prone to introduce defects into the software components). We supposed that the TCNs are particularly suitable to address the JIT-SDP (software defect prediction) problem that is characterized by a huge amount of data (extracted at the commit level) organized as multivariate time-series. TCN can look very far into the past to make a prediction using a combination of deep networks (augmented with residual layers) and dilated convolutions. The underlying hypothesis is that since TCN is characterized by casualness in the convolution architecture design and sequence length [8], they are more suitable for the software defect prediction scenario. The reason lies in the need to learn, during software evolution analysis, causal relationships between quality and process metrics and the presence of defects. Moreover, the basic architecture of TCN is customized by adding a hierarchy of attention layers able to capture both low and high frequency change patterns of metrics time series over time. Finally, the adoption of TCN is also suitable given the availability of a huge number of data for several systems (since data extraction is performed at the commit level).

Summarizing, if compared to other JIT models proposed in the literature, our model is characterized by the following novelties:

- it is based on a mix of source code quality and process metrics (instead of using Mockus and Weiss's model) that is suitable to measure the quality of both the source code elements and of the activities performed around each commits; our findings show that the proposed metrics are effective in predicting the probability of changes inducing a fix;
- fine-grained analysis for both time and space: using historical data gathered from the repository, it can predict specific classes (instead of components) as defect-prone in the next commit (instead of releases);
- exploits a variant of TCN adding a hierarchy of attention layers to better capture relationships of product and process metrics over time.

The performance comparison (for both effectiveness and efficiency) between the proposed approach and the existing JIT models are discussed in the results.

This work extends the preliminary study proposed in [4] adding the following contributions:

- it exploits a set of ad-hoc process metrics. The experiments highlight that extending the features model with the proposed process metrics improves final end-to-end prediction;
- more in-depth experimentation discussing four research questions that cover:
  - performance assessment of the TCN on both the original and the extended model;
  - comparison with other neural networks architectures (LSTM, CNN-1D, CNN-2D);
  - comparison with the most relevant approaches in the literature;
  - a study of how buggy revisions ratio (i.e., imbalance) impacts final end-to-end prediction performance;
- a higher number of systems (six instead of four) including two larger systems for size (i.e., LOC and number of classes). For all systems also the history size is extended, studying a higher number of commits related to a longer time frame.

The paper is structured into eight sections. Section 2, reports a description and a comparison with the main related work. Section 3 contains some background information while a discussion of the proposed approach is reported in Sect. 4. Section 5 describes the experiment carried out whose results are reported and discussed in Sect. 6. Threats to validity are discussed in Sect. 7 and, finally, the conclusions are drawn in Sect. 8.

2 Related work

Several works applied deep-learning techniques to boost defect prediction performances at change-time. In [70], the authors applied the deep belief network for the first time to the just-in-time software defect prediction. The authors evaluated their approach on datasets taken from 6 large open-source software projects achieving average recall and F1-score of 69% and 45%. Besides, for cost-effectiveness, this approach can identify over 50% defective changes by reviewing only 20% of lines of code. In a recent work [54], authors proposed a prediction model based on graphs representing program execution flows and deep neural networks for automatically learning defect features. Since control flow graphs are built from the assembly instructions the whole source code is first compiled. This implies that this model applies only when the source code is available and compilable. In some studies [26, 66], DBN and LSTM have been used to extract features from the project's source code while in another work [68], a deep neural network with a new hybrid loss function is used to train a DNN learning top-level feature representation. The extensive empirical investigation carried out on a benchmark data-set

with 27 defect data demonstrates the superiority of the proposed approach in detecting defective modules when compared with 27 baseline methods. In [44], a genetic algorithm is used as a feature optimization model selecting a set of features as input of a DNN. The authors evaluated this model on five projects belonging to the well-known PROMISE data-set obtaining the best accuracy in the literature (98%), to the best of our knowledge. Anyway, at least two important limits are necessary to highlight: firstly, it is not possible to explore the source code, secondly, the contextual data are not comprehensive (e.g., no data on maturity are available).

Recently research on JIT techniques applied to SDP has increased rapidly. All the JIT models are designed based on the assumption that past code change properties are similar to future ones or, in other terms, that "the predictive power of JIT models does not change by time" [38]. To identify the defect-inducing changes in [41] authors proposed a prediction model based on JIT quality assurance. Later on, the authors carried out an empirical investigation on 11 open source projects to evaluate how JIT models perform in the context of cross-project defect prediction [40]. Their main findings report that JIT models learned using other projects are a viable solution for projects with limited historical data and only when the data used to learn them are carefully selected.

In [69] Yang et al., still using the same basic change features used by Kamei et al's [40], uses a combination of data preprocessing and a two-layer ensemble of decision trees to improve the performance of JIT defect prediction. In the inner layer, they combine Decision Tree and Bagging to build a Random Forest while in the outer layer they use random under-sampling to train many different Random Forest models and ensemble them once more using stacking. The results of the empirical investigation carried out on data of six open-source projects, show that on average across the six datasets, the approach achieves an average F1-score of close to 50%.

Afterward, the authors replicated their study in [72]. The goal of replication was to verify the results of the original study [70] and investigate whether diversifying the set of classifiers, optimizing the weights of the classifiers when combining them and additional layers in the ensemble, can enhance performance in predicting defects. For this reason, the authors applied a new deep ensemble approach assessing the depth of the original study and achieving that the F1 score is statistically significantly higher at an alpha of 0.05 for the new approach across all projects compared to the original approach.

Chen et al. [22], believing that supervised methods should have better prediction performance, designed a new supervised method for JIT-SDP. This method applied a multi-objective optimization algorithm to software defect prediction. Experimental results, carried out on six open-source projects, show that the proposed method is superior to all state-of-the-art prediction models for Accuracy and P_{opt} (the normalized version of the effort-aware performance indicator). They also found, for example, that the proposed method for F1, average value, can obtain 24%, 45%, 18%, and 9% on average when compared to the supervised methods that gave the best results.

Pascarella et al. [52] proposed a fine-grained prediction model, based on the Random Forest technique, to predict commit by commit the specific file that is defective. The empirical investigation shows that 43% of defective commits are mixed by buggy and clean resources, obtaining a maximum value of 71% for the F-measure and stable performance across the considered projects. It is important to note that authors used 24 basic features representing a modified version, to work at file-level in a commit, of those previously proposed by Kamei et al. [41] and Rahman and Devanbu [57].

Hoang et al. [35] proposed a prediction model based on Convolutional Neural Network, whose features were extracted from both commit messages and code changes. Empirical results show that the proposed approach, evaluated in three different settings: cross-validation, short-term and long-term, achieved improvements of 8.96%, 7.00%, and 8.05% in terms of AUC compared to the best baseline. Results also indicate that code changes are more important to detect buggy commits than commit messages.

Cabral et al. [20] propose a novel class imbalance evolution approach for the specific context of JIT-SDP. Their approach managed to produce up to 63.59% more balanced recalls on the defect-inducing and clean classes than state-of-the-art class imbalance evolution approaches.

In [27] authors propose a new software defect prediction method based on the improved histogram-based isolation forest able to mitigate some side-effects caused by the imbalanced training dataset and enhance prediction performance. The authors experimented on ten imbalanced NASA software defect datasets and to validate the effectiveness of the proposed method make comparisons with traditional ensemble learning methods such as Boosting, Bagging, and Random Forest. The best performances of the F-measure are under the 87% value.

To date, all the JIT models are based on the software prediction model introduced by Mockus and Weiss [48]. This model is made up of a set of 14 software changes, grouped in five dimensions, whose effectiveness to predict the introduction of a defect was empirically shown by authors [48] and in [41] where authors validated it through a "large-scale study of six open-source and five commercial projects from multiple domains" [41].

Concerning the described approaches, our study proposes a JIT defect-prediction technique using different

metrics as features and a different neural model, based on a TCN variant, as a predictor. It allows the developers to fix and check the defects just at the time they are introduced achieving the following advantages:

- predicting where defects will be located in focused code components provides large effort savings over coarser-grained predictions;
- predicting defect ahead of the time help to focus the testing effort on the right part of the system saving resources;
- only properties related to change are considered for predictions. Therefore, the prediction can be performed immediately ensuring that the design decisions are still in the minds of developers.

3 Background

This section provides the essential notions needed to follow the deep neural networks based approach description.

3.1 Just-in-time software defect prediction

The existing techniques to evaluate the defectiveness of software can be categorized into long-term predictions and short-term predictions. Long-term techniques pertain to models able to analyze the historical data stored in software releases to identify the artifacts that are more prone to defect in future. These techniques were applied using Object-Oriented metrics [24] or process metrics (e.g., the entropy of changes [33]). The main limitation of the long-term prediction models is that they do not provide developers with immediate feedback. To overcome this limitation, short-in-time prediction models exploit the characteristics of a commit to perform at check-in time predictions of the likelihood of a commit introducing a defect. These models offer at least three advantages: first, the granularity of predictions is fine, developers need to check only code modified by commit to locate the defective code; second, the programmer who made the commit still keep in mind the details of the development and this can increase the efficiency of defects detection and repair; third, the expert to perform code checking on the defect-prone modules is known and immediately available. Main short-term prediction models focused on just-in-time quality assurance techniques. This technique [41] tries to reduce the effort of a reviewer focusing on "identifying defect-prone ('risky') software changes instead of files or packages". In this way "developers can review and test these risky changes while they are still fresh in their minds" [41]. Later on, authors of [70] and [9] proposed the usage of alternative techniques for just-in-time quality assurance, such as cached history, deep learning, and textual analysis, reporting promising results. Other approaches proposed the use of deep-learning [70], textual analysis [9], and unsupervised methodologies [71].

3.2 Machine learning and deep learning algorithms

Machine learning (ML) is the study of algorithms that allow computer programs to automatically improve through experience [47]. One of the most important factors for the performance of machine learning methods concerns data representation (or features) on which they are applied. At present, among the various ways of learning representations, the most interesting ones are the Deep Learning (DL) methods. These methods are formed by the composition of multiple nonlinear transformations, to yield more abstract and more useful representations [12]. DL takes inspiration, by the way, biological nervous systems process information (artificial perceptrons can be seen as human neurons).

The architecture of the DL methods consists of constructing multiple levels of representation or learning a hierarchy of features. The depth of a circuit is the length of the longest path from an input node of the circuit to an output node of the circuit. The crucial property of a deep circuit is that its number of paths, i.e., ways to re-use different parts, can grow exponentially with its depth. DL promotes the re-use of features and leads to progressively more abstract features at higher layers of representations (more removed from the data). DL has been successfully applied in the complex domain since it allows us to gradually improve overall performance by adopting black-box models. DL applications are increasingly growing in the fields of speech recognition, image recognition, bioinformatics, financial fraud detection, computer vision, natural language processing, health informatics, audio recognition, social network filtering, and machine translation.

Among networks, deep neural networks consist of several hidden layers where each layer represents hierarchies of concepts used to perform pattern classification and feature learning. Similarly to the training of a traditional neural network, for the DL network, the forward and backward phases are expected. In the forward step, the nodes' activation signals flow from the input to the output layer. In the backward step, the biases and the weights are eventually adjusted to improve the network performance.

In this study, we adopt TCN networks, a class of deep neural networks, and propose a variant of them. Moreover, we compare the performance obtained by using the proposed TCN network with the ones obtained using other alternative neural networks: LSTM, CNN-2D.

3.2.1 LSTM networks

Long Short-Term Memory (LSTM) networks [61] are a type of Recurrent Neural Network (RNN) architectures used in the field of DL.

A common LSTM unit includes a cell, an input gate, an output gate and a forget gate. LSTM networks are able of learning over long time sequences and retaining memory [36] because the cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. Therefore, the application of LSTM as a prediction task in the area of software defect prediction allows looking back to the history of fine-grained quality metrics.

The LSTM classifier architecture is described in [5]. It is made up of three types of layers: the input, the hidden, and the output layers.

The input layer is a prototype bringing the data into the network for further processing. It requires a 3D tensor with shape: number of samples, number of time steps, and number of features. Next, the hidden layer seeks to construct the relation between the input and the output. It represents the principal part of the network and can contain one single or multiple layers. LSTM units return all of the outputs from the unrolled LSTM units through time allowing learning sequences of observations and well-suiting to time series problems. To prevent the overfitting problem the dropout regularization method is used for every LSTM layer to improve the model performance. Finally, the output layer is used as a prototype between the network and the output. It contains a feed-forward neural network that is a regular fully connected layer and allows transforming the 3D tensor at the hidden layer output to a 1D array at the classifier output. The LSTM training requires the definition of the objective function as "categorical cross-entropy", a logarithmic loss function specially used to solve the multiple mutually-exclusive class problem.

This function returns the cross-entropy H(p, q) between a predicted probability distribution (p(x)) and a true probability distribution (q(x)). It is given by:

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

3.2.2 CNN

A Convolutional Neural Network (CNN) is based on a feed-forward architecture consisting of several stages, each specializing different functionalities [2].

In particular, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth; where neurons belonging to one layer are connected only to a small region of the layer before it, rather than to all neurons in a fully connected way. The architecture of CNN is shown in Fig. 1. Each CNN is made up of a chain of ordered blocks, where each block identifies one level of the network that transforms one volume of activations into another through a differentiable function. It consists of the following levels:

- **Input level**: represents the set of data in the form of numbers to be analyzed.
- **Convolutional level** (Conv): extracts the main features through the use of filters.
- **ReLU level** (Rectified Linear Units): introduces non-linearity to a system that is essentially calculating linear operations during convolutional levels, through the scalar product between the filter and the receptive field. It cancels all negative values, increasing the nonlinear properties of the model and the global network without affecting the receptive fields of the convolutional level. The ReLU level applies the function $f(x) = \max(0, x)$ to all values in the input volume.
- **Pool level**: allows identifying if the study characteristic is present in the previous level and drastically reduces both height and width, that is the spatial dimension, of

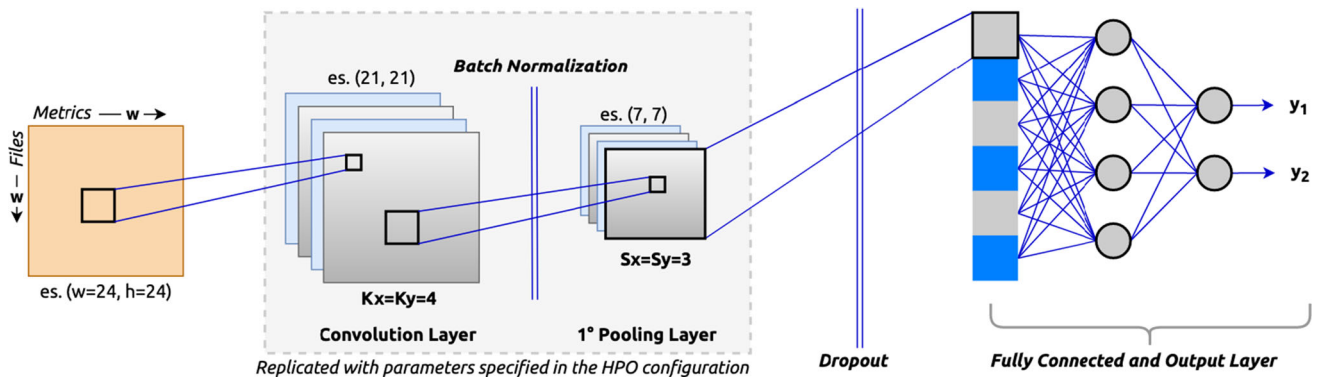

Fig. 1 Convolutional neural network

the input volume, and also the computational requirements for future levels.
– **FC level** (Fully Connected level): connects, first, all the neurons of the previous level to establish the various identification classes displayed in the previous levels according to a certain probability and, then, performs the classification. It takes an input volume (whatever the output of the convolutional level or of the ReLU or of the pool level that precedes it) and generates a vector whose dimension is N that is the number of classes from which the program must choose.

There are CNNs of 1, 2, or 3 dimensions. These CNNs work always, in the same way, the only differences are the structure of the input data and how the filter, also known as a convolution kernel, moves across the data. In our work, we have considered CNN-2D because it is suitable for our data structure.

With a 2D convolution level, a 3×3 convolution window contains $3 \times 3 = 9$ feature vectors. With the 1D convolution level, a size 3 window contains only 3 feature vectors making convolution windows of bigger sizes a viable choice.

3.2.3 Temporal convolutional networks (TCN)

The temporal convolutional network (TCN) is a type of network with two distinguishing characteristics: 1) the convolutions in the architecture are causal to ensure that only the current and passing samples are considered in the evaluation of the output at a given instant of time; 2) the architecture can take a sequence of any length and map it to an output sequence of the same length. Beyond this, it is important to highlight the ability for the TCNs to look very far into the past to make a prediction using a combination of very deep networks (augmented with residual layers) and dilated convolutions.

The TCN architecture to perform the classification uses the last sequential activation of the last layer because it can summarize in a single vector the information extracted from the complete input sequence.

4 Approach

In this section, the proposed software defect prediction approach is described focusing on the following aspects:

– an improved granularity of the prediction in space and time;
– the set of fine-grained metrics exploited as features;
– the adopted deep neural network architecture.

Concerning the granularity, the proposed approach aims at predicting the software defectiveness at a class level across software project commit events. The analysis at the commit-level makes available a greater amount of training data and allows leveraging deep learning approaches.

Time granularity is equally important. Indeed, the sampling at each release makes the prediction less useful in practice. Developers need just-in-time information on defect-proneness to react quickly directing the testing effort toward the components that are most likely to exhibit defects. In addition, coarse-grained predictions usually increase the effort spent by developers in locating the software defects within large files or components.

Looking at the selected features, two classes of metrics are largely used to predict software defects:

– product metrics, describing the source code internal structure quality;
– process metrics, modeling the development process and developers interactions quality.

The proposed approach adopts a mix of these metrics.

Another critical aspect is the prediction technique to adopt. Looking at existing literature, deep learning techniques have been exploited for the tasks using neural machine translation approaches but the results are still not satisfying. Given the higher volume of data available using commit level analysis, the goal of this work is to investigate to what extent a just-in-time metric-based approach is effective in predicting software defects and which neural network architecture is the best choice for the task. From the architectural point of view, we propose a variant of Temporal Convolutional Networks. Since they imply casualness in the architecture design [8], they are suitable to our prediction problem where the causal relationships among the metrics evolution and software defect presence needs to be learned. The proposed variant includes a hierarchy of attention layers more efficient to capture the complex relationships of product and process metrics over time.

4.1 The feature model

One of the most commonly used metrics suites is that proposed by Chidamber and Kemerer (CK) [24]. It originally consisted of 6 metrics that were later extended by adding missing aspects. In particular, Abreu [19] introduced a set of metrics called Method Hiding Factor (MHF) to measure the information hiding aspects of a class. Several studies discussed the strong correlation of these metrics to software faults [18, 62]. For this reason, we also included these metrics in the features model used to train our classifiers.

Moreover, several additional factors contribute to introduce software defects. For example, the development process followed by developers can strongly influence software defectiveness. This allows us to introduce the process metrics that explore the kind of performed changes and the characteristics of authors and committers involved in the changes. The process metrics considered in this study are:

- **Developer Seniority (SEN)**: It measures the seniority of the software developer in days, at each commit. We calculated its author' seniority as the difference with the commit date of its first commits in the repository using the following formula:

$$\mathbf{SEN}(\boldsymbol{c_j},\boldsymbol{d_i}) = \mathrm{Cd}(c_\mathrm{j}) - \mathrm{Fc}(d_\mathrm{i})$$

  where:

  $\boldsymbol{c_j}$: the commit for which we want to evaluate the author's seniority.

  $\boldsymbol{d_i}$: the developer who authored the commit c_j.

  $Cd(c_j)$: the date of the commit c_j.

  $\mathrm{Fc}(d_i)$: the date of the first commit authored in the source code repository by developer d_i.
- **Owned Commit (OC)**: It is based on the definition of code ownership [17] which considers a *file owner* as a committer that performed, a given percentage of the total number of occurred commits on the file.

  In our context, we consider as owners the developers that collectively performed at least half of the total changes on that file. This assumption is in line with the literature [31] that evaluates the ownership as the percentage of changes of the contributor with the highest number of changes.

  More formally, we define the set of *owners of the file* f_j *at commit* c_k as follows:

$$O(f_j, c_k) = \{o_1, \ldots, o_I\} \tag{1}$$

  The set contains the committers that, overall, performed the 50% of the changes on f_j in the commits interval $[c_s, \ldots, c_k]$ satisfying the following condition:

$$\sum_{i=1}^{|O(f_j,c_k)|} R(o_i,f_j,c_k) \geq 0.5 \cdot T_c(f_j,c_k) \tag{2}$$

  where c_s is the starting point of our period of observation and $R(o_i,f_j,c_k)$ is the number of changes performed by developer o_i on file f_j in the time interval $[c_s,c_k]$, and $T_c(f_j,c_k)$ is the total number of changes performed on f_j in the time interval $[c_s,c_k]$ by all developers that worked on it. To compute the set $O(f_j,c_k)$, we build an ascending sorted list of committers according to the number of changes performed on f_j in the time interval $[c_s,c_k]$, and select the minimum set of topmost committers in the ranked list that satisfies the above constraint. With the above definition, it is possible to define the *Owned Commit* (OC) binary predicate as follows. Let be $O(f_j,c_k)$ the set of owners of file f_j at commit c_k. OC indicates if the developer d_i authoring the changes on f_j at commit c_k is one of the owners of the file and is defined as follows:

$$\mathrm{OC}(d_i,f_j,c_k) = d_i \in O(f_j,c_k)$$

- **Number of File Owners (NFOWN)**: Given the definitions above, this can be defined, for a file f_j and a commit c_k, as the cardinality of the set $O(f_j,c_k)$:

$$NFOWN(f_j,c_k) = |O(f_j,c_k)|$$

- **Owned File Ratio (OFR)**: for a developer d_i, a file f_j and a commit c_k, this is the ratio $R(d_i,f_j,c_k)$ of changes performed by d_i with respect to the total changes performed by all developers on file f_j from the start of the observation period (i.e., in the commits interval $[c_s,\ldots,c_k]$).

The last three metrics are file or developer properties and can be easily computed using VCS logs. The **Time since last commit (TSLC)**, for a given file f_j and a commit c_k, is the number of days passed since last commit on f_j. For a given developer d_i, the **Commit Frequency (CF)** is the number of commits by month authored by d_i whereas the **Mean time between commits (MTBC)** is the average time (in days) between all subsequent commits authored by d_i. The complete features set are a combination of the above product and process metrics as described in Table 1. The table reports each metric in a row along with an acronym and a short description (last column).

The process deployed for the feature extraction and training/test datasets generation is described in Fig. 2a. The process starts when all the commits logs are gathered by the VCS to evaluate the process metrics. From the VCS also the CK and MOOD metrics are evaluated. However, for each commit, the corresponding source code has been downloaded and analyzed to check the evolution of the CK and MOOD metrics of Table 1.

The open-source tool JaSoMe[1] (Java Source Metrics) is used to evaluate metrics. It is an open-source tool to mine internal quality metrics from systems source code and does not require source code to be compiled. The metrics are calculated also by using other existing tools [3, 60] to ensure the correctness of the obtained results. These metrics are then cleaned (incomplete and wrong samples are removed) and normalized (min-max normalization).

Similarly, all information about the bugs (issue logs) is extracted by the tracking system (BTS). In particular, for

[1] https://github.com/rodhilton/jasome/.

Table 1 The Metrics included in the prop features model

Type	Acronym	Description
Product metrics (CK and MOOD)	Number of Attributes Inherited (Ai)	Attributes inherited but not overridden
	Number of Attributes Defined (Ad)	Attributes defined within class
	Number of Attributes Overridden (Ao)	Attributes in class that override an otherwise-inherited attributes
	Number of Attributes Inherited Total (Ait)	Attributes inherited overall
	Class Relative System Complexity (ClRCi)	avg(Ci) over all methods in class
	Number of Public Attributes Defined (Av)	Number of defined attributes that are public
	Depth of Inheritance Tree (DIT)	The maximum depth of the inheritance hierarchy for a class
	Class Total System Complexity (ClTCi)	sum(Ci) over all methods in class
	Number of Hidden Methods Inherited (HMi)	Number of inherited (but not overridden) methods that are non-public
	Number of Hidden Methods Defined (HMd)	Number of defined methods that are non-public
	Lack of Cohesion in Methods (LCOM*)	It indicates whether a class represents a single abstraction or multiple abstractions
	Number of Methods Inherited Total (Mit)	Methods inherited overall
	Number of Methods Defined (Md)	Methods defined within class
	Number of Methods Inherited (Mi)	Methods inherited but not overridden
	Number of Methods Overridden (Mo)	Methods in class that override an otherwise-inherited method
	Number of Attributes (NF)	The number of fields/attributes
	Number of Methods (NM)	The number of methods
	Number of Methods Added to Inheritance (NMA)	The number of methods a class inherits adds to the inheritance hierarchy
	Number of Inherited Methods (NMI)	The number of methods a class inherits from parent classes
	Number of Ancestors (NOA)	Total number of classes that have this class as a descendant
	Number of Children (NOCh)	Number of classes that directly extend this class
	Number of Descendants (NOD)	Total number of classes that have this class as an ancestor
	Number of Links (NOL)	Number of links between a class and all others
	Number of Parents (NOPa)	Number of classes that this class directly extends
	Number of Public Attributes (NPF)	The number of public attributes
	Number of Static Attributes (NSF)	The number of static attributes
	Number of Static Methods (NSM)	The number of static methods
	Polymorphism Factor (PF)	PF measures the degree of method overriding in the class inheritance tree
	Number of Public Methods Defined (PMd)	Number of defined methods that are public
	Raw Total Lines of Code (RTLOC)	The actual number of lines of code in a class
	Specialization Index (SIX)	How specialized a class is, defined as (DIT * NORM) / NOM;
	Total Lines of Code (TLOC)	The total number of lines of code, ignoring comments, whitespace
	Inheritance Factor (MIF)	Mi / Ma
	Method Hiding Factor (MHF)	PMd / Md
	Number of Methods (All) (Ma)	Methods that can be invoked on a class (inherited, overridden, defined as Ma = Md + Mi
	Weighed Methods per Class (WMC)	The sum of all of the cyclomatic complexities of all methods on a class
Process metrics	Commit Frequency (CF)	The authoring frequency of the developer authoring the change
	Developer Seniority (SEN)	The seniority of the developer authoring the changes
	Number of File Owners (NFOWN)	The total number of owners of the changed file
	Owned Commit (OC)	A predicate indicating if an author is among the file owners or not.
	Owned File Ratio (OFR)	The ratio of file lines owned by the developer authoring the change.
	Time since last commit (TSLC)	The days passed since last authored commit
	Mean time between commits (MTBC)	The mean time between two commits of the developer authoring the commit

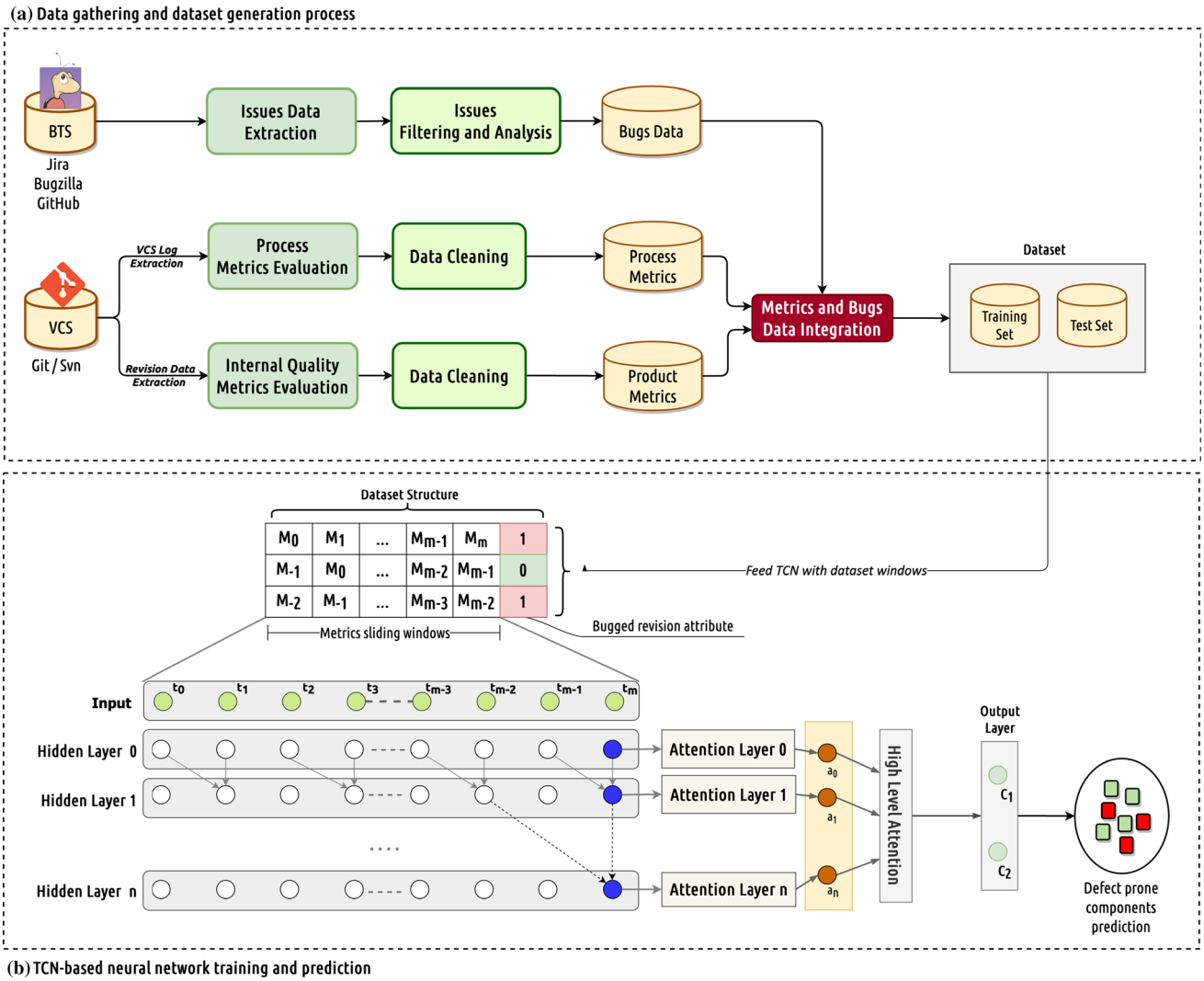

Fig. 2 Overall process and classifier architecture

each class, the number of commits and bugs is evaluated from the log gathered from the BTS repository. For each commit, the extracted information regards the changed files (i.e., their names and the total number) and the commits (i.e., their ID, timestamp, commit parent, and commit note). This information allows identifying each change that induces a fix. To this aim, an approach inspired to [29] is used: for each ID registered in the BTS of the analyzed project, the corresponding changes (basing on the matching of the commit note) are selected. The extracted issues are then classified on their type attribute (issues analysis and filtering) and the issues that are not classified as bug fixes (e.g., improvement, enhancements, feature additions, and refactoring tasks) are filtered out.

The traceability between the VCS and the BTS repositories is ensured by the issue ID. This allows identifying, for each class, all the issues that are related to bug fixes since they are used to tag the faulty revisions associated with each class. Moreover, the issues are selected by their status: the *CLOSED* and the *DONE* statuses are considered. However, *CLOSED* and the *DONE* statuses suggest that the changes are committed in the repository and applied to the components (a commit is performed). In this way, the bugged classes for each commit registered in the VCS repository can be identified. The final dataset, for each class of the system, contains the evolution, by commits, of the calculated process and product metrics integrated with bug presence information (bugs data). The integrated metrics and bugs data are then used to build the training and testing datasets.

4.2 Deep neural network-based classifiers

The classifier is based on a TCN network that is compared, for what concerns the obtained performances, with different deep neural network architectures: LSTM, CNN. The

classification is performed in three steps: (i) data samples pre-processing, (ii) training, and (iii) testing.

The first step allowed cleaning the raw dataset: the incomplete and wrong sampled data sessions are removed and the attributes are normalizing using a min-max normalization approach. The second step allowed defining a set of labeled traces (the structure of the traces depends on the kind of considered network).

The starting point is the VCS direct acyclic graph (DAG) integrated with data extracted from the BTS system using information extracted from VCS logs. The linearization algorithm performs a graph traversal to generate, for each class in each branch, a commit sequence where commits are marked with bugs opening and fixing events. This step generates a set of commit windows that are used to drive metrics and label calculation. Figure 3 shows a small running example clarifying how training windows are generated. The process starts in Fig. 3a where a small DAG made of seven commits (from C_1 to C_7) belonging to two branches (i.e., master and B1) is shown. This DAG reports the evolution of four files over time. Let us focus on the Server class for which a bug is opened in C_3, on the master branch, and fixed in C_4, on the B1 branch. The traversal produces, for the Server class, two linear windows (W_1 and W_2) one for each branch in which the class is present including information on when the bug is open and, later, closed. Looking at the exemplified scenario, the Server bug affects just one commit on the B1 branch as reported in window W_2 (since it is fixed in the fork at C_4). However, the same bug affects two commits on the master branch since the fix is not applied until the merge commit at C_7. The linearization is a determining step since allows to follow opening and fixing along the DAG paths, propagating information on where a bug is alive, for a given class, onto the linear windows used for training. Figure 3c reports, for each window W_1 and W_2, the sequence of metrics vectors evaluated for each commit of the window and associated to the correct label that indicates bug presence at that commit. These windows are collected and used to generate training and test data.

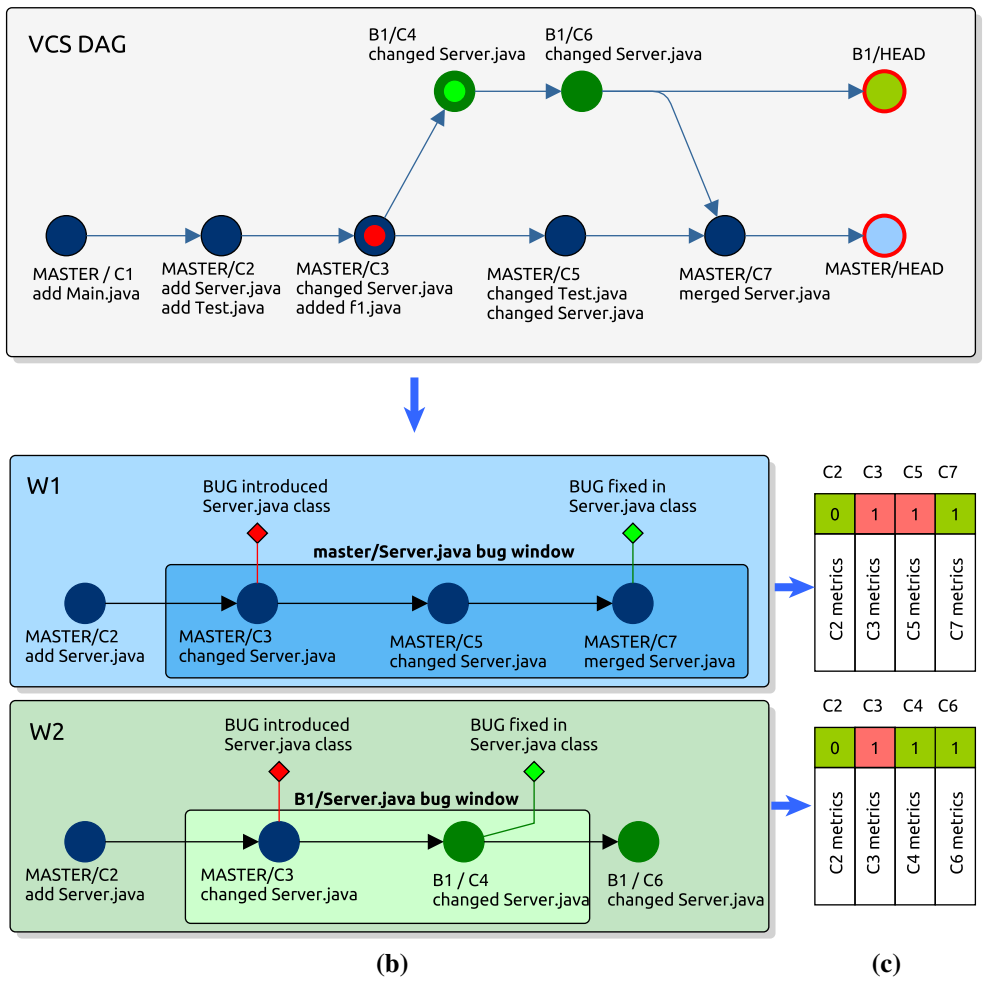

Fig. 3 A small running example showing training and test windows generation

Specifically, to perform validation during the training step, nested cross-validation is used [65].

It exploits an outer loop and an inner loop respectively for error estimation and for parameter tuning. The inner loop splits the training set into a training subset (the model is trained on this subset) and a validation set (the parameters that minimize error on the validation set are selected). In the outer loop, the dataset is divided into multiple different training and test sets, on each split the average error is computed to obtain a robust estimate of model error. The evaluation of the trained classifier is carried out using real data that is never seen before by the classifier.

The classifiers proposed in this paper are constructed by using the python deep learning library, Keras[2]. In the following, more details on how training is performed are provided for each classifier.

4.2.1 Temporal convolutional network with hierarchical attention layers

In the proposed TCN, we added a hierarchical attention mechanism through the network levels introduced by [71], because we believed that the representation just explained could be too reductive for complex relationships, such as those present in multivariate time series of quality metrics internal bugs.

The following types of layers are considered:

- **Input level**: it is the neural network entry point and includes a node for each set of considered features at a given time;
- **Hidden layers**: they include the artificial neurons. The output of each neuron can be calculated as a weighted sum of its inputs and can be transmitted through an activation function (in this study Mish, Swish, and ReLu functions are used) or through a soft-plus function;
- **Attention layers**: they model (both in input and in output) the relationships regardless of their distance;
- **Batch normalization**: it allows to improve the training performance of deep feed-forward neural networks;
- **Layer output**: it gives the required output.

The proposed TCN architecture can be parametrized by the number of hidden layers, all having the same length as the input layer. To impose the layers' length coherence, suitable padding of length (kernel size-1) is used. This architecture ensures that at each evaluation, the output is calculated looking only at current or previous samples. The network exploits dilated convolutions allowing an exponentially wide receptive field that depends on a dilation factor d_f between every two adjacent filters (similar to a fixed step). As the layer number increases, the dilation factor raises at an exponential rate. When the kernel size becomes k_l, the data enabled at the lower layer is $(k_l - 1)d$ and still rises at an exponential rate by the number of the network layers. The prediction is performed on the last sequential activation of the last layer (i.e., the output layer) which summarizes data from the input time series into a single feature vector that is used to generate the final output.

However, this model can be unable to mine the high number of complex and long-living relationships among metrics and issues. For this reason, a hierarchical attention approach [71] is used adding two levels of hidden layers across the network as already done in [15, 16]. Attention layers help to learn the relationships among items regardless of their remoteness in the input or output sequences.

This TCN-based architecture is depicted in Fig. 2b for n hidden layers, the matrix of weights $\mathbf{L}_i \in \mathbb{R}^{K\times m}$ defined as:

$$\mathbf{L}_i = [\mathbf{l}_1^i, ..., \mathbf{l}_m^i], \tag{3}$$

where K is the filters' number at each layer and m is the length of the window and i is the number of the convolutional activations layer (where $i = 1, \ldots, n$).

Moreover, we can define the layer attention weight $\mathbf{m}_i \in \mathbb{R}^{1\times m}$ as:

$$\mathbf{m}_i = \text{softmax}(\tanh(\mathbf{w}_i^T \mathbf{L}_i)) \tag{4}$$

where $\mathbf{w}_i \in \mathbb{R}^{K\times 1}$ are the trainable parameter vectors. For the layer i, the corresponding set of convolutional activations is computed as $\mathbf{a_i} \in \mathbb{R}^{K\times 1} = f(\mathbf{L}_i \boldsymbol{\beta}_i^T)$ where $f(\cdot)$ is one activation function among ReLU, Mish and Swish [46] and $\boldsymbol{\beta}_i$ are the weights of the attention layer. Finally, the convolutional activations $\mathbf{A} \in \mathbb{R}^{K\times n} = [\mathbf{a}_1, ..., \mathbf{a}_i, ..., \mathbf{a}_n]$ of the hidden layers allow to compute the representation of the last sequence to ensure the final classification:

$$\boldsymbol{\alpha} = \text{softmax}(\tanh(\boldsymbol{\omega}^T \mathbf{A})) \tag{5}$$

$$\mathbf{y} = f(\mathbf{A}\boldsymbol{\alpha}^T) \tag{6}$$

where $\boldsymbol{\omega} \in \mathbb{R}^{K\times 1}$ and $\boldsymbol{\alpha} \in \mathbb{R}^{1\times n}$ are respectively the vector of weights and the output of the high-level attention layer, and $\mathbf{y} \in \mathbb{R}^{K\times 1}$ is the neural network final output. Finally, a batch normalization layer [37] is included to enhance the training process.

Figure 2b also shows the data used by the network in the training step. The dataset is composed of a set of labeled traces $T_r = (M, l)$.

Each system development instant is represented as a row M and is associated with a binary label l that allows predicting the presence of design smell. For each, a vector V_f function is sent to the classifier during the training phase. In the same way, data has been used by the input layer of

[2] https://keras.io.

the LSTM networks. For both the adopted networks (TCN and LSTM), it is important to carefully select the time series sliding window size hyperparameter, the number of layers that depends on the duration of the dynamics that the neural network must learn from data.

4.2.2 Convolutional neural networks training

Figure 1 describes the architectures of CNN models used to predict software defects. Inspiring to typical CNN architecture applied in image classification, it includes a feature extraction subnet that is followed by a classification subnet. The first subnet is made up of a sequence of hidden layers (convolution, batch normalization, and max-pooling layers) with variable lengths where sizes and numbers as driven by the hyper-parameter optimization process. The batch normalization reduces the effects of different input distributions across training mini-batches, optimizing the entire training process allowing convergence for deeper networks. Following the last max-pooling layer, the dropout is also executed to avoid over-fitting. The figure also shows that to drive CNN 2D, metrics data needs to be reshaped. In particular, the evaluated metrics vectors are organized, for each commit, as a matrix that needs to be dimensioned. This allows ensuring sufficient space for the systems under study and the features set adopted.

5 Experiment description

The experimentation is conducted by using the approach described in Sect. 4 on a dataset composed of six open-source systems.

In the following, the research questions, the dataset description and the experiment settings are reported.

5.1 Research questions

The goal of the experiment is to study the performance of the proposed approach for continuous just-in-time defects prediction in the context of open-source software projects. Specifically, we investigated the following research questions:

RQ_1: *Is the proposed TCN-based approach effective for just-in-time defect prediction?* This research question aims to evaluate. the F1 measure of the proposed TCN variant on the described set of features, to predict the presence of software defects for each investigated system. These measures are compared with the best results present in the literature.

RQ_2: *Is the proposed TCN variant more effective than other deep neural network architectures used in the literature for just-in-time defect prediction?* This research question aims to compare, for each investigated system, the accuracy and F1 measure of the proposed TCN variant to predict the presence of software defects, with those obtained from different permutations of alternative networks such as LSTM, CNN-2D.

RQ_3: *Does the proposed approach provide a performance benefit from both product and process metrics? What is the impact of the newly proposed process metrics?* This research question aims to assess the value of each family of features independently from the others. To answer this question, we build two different software prediction models relying on (i) only product metrics, (ii) both product and process metrics.

RQ_4: *What is the effect of imbalanced classes on the performances of the TCN-based approach to predict buggy Vs not buggy changes when trained on the proposed metrics?* This research question aims to analyze the effect of imbalanced data on the performances, evaluated in terms of F-measure, of the TCN-based approach for just-in-time software defects prediction using both product and process metrics.

5.2 Dataset

In this study, we used datasets from six open-source projects. In order to perform projects sampling for our study, we adopted criteria widely recognized in the literature [25, 55]. Specifically, we queried GitHub enforcing the following criteria:

1. the programming language, coherently with the product metrics used, is Object-Oriented (in all cases it is Java);
2. the Git repository is neither archived nor private;
3. the history stored on the Git repository, the minimum history among the considered systems is 20 releases and more than 1000 commits (filtering trivial projects);
4. the projects differ in application domains, sizes, revisions (number of files);
5. the history spans over a large time frame, at least 8 years long;
6. the number of committers must be significant ($\geq$ 20).

We obtained a list of thousand systems sorted by decreasing number of community adoption metrics (i.e., number of forks, number of watches and number of stars).

Table 2 reports the names of the final selected projects (first row), the dates of the first and the last considered commit (second row), the total number of commits for each system (third row), and the number of revisions for each system (fourth row). Finally, in the last row for each system is reported the number of buggy revisions.

Table 2 Characteristics of the considered systems

Systems	ZooKeeper[3]	Xerces2 Java[4]	JFreeChart[5]	Jackson Data Format[6]	Jackson Core[7]	Commons Imaging[8]
from-to dates	Nov 2007—Mar 2020	Nov 1999—Mar 2020	Jun 2007—Feb 2020	Jan 2011—Mar 2020	Dec 2011—Feb 2020	Oct 2007—Feb 2020
Revisions	1,264,735	486,371	403,467	253,176	3,058,530	3,684,838
Commits	2101	5510	3786	1901	2103	1233
Buggy revisions	105,210	10,514	31,752	19,383	137,074	812,263

[3] https://github.com/apache/zookeeper

[4] https://github.com/apache/xerces2-j

[5] https://github.com/jfree/jfreechart

[6] https://github.com/FasterXML/jackson-dataformat-xml

[7] https://github.com/FasterXML/jackson-core

[8] https://github.com/apache/commons-imaging

Table 3 Overview of the evaluated hyper-parameters

CBCEFB hyperparameter	Considered values
Network size	Small, Medium, Large
Optimization algorithm	SGD, Adam, RmsProp, Nadam, Adamax, Adagrad
Activation function	Mish, ReLu, Swish
Learning rate	[0.05, 0.015]
Dropout rate	[0.10, 0.25]
Batch size	{ 16, 32, 64, 128, 256, 512 }
Number of layers	{ 5, 6, 7, 8, 9 }
Sliding Window Size (TCN/LSTM)	{ 16, 32 }

5.3 Experimental setting

For each research question, one or more experiments are conducted. Starting from RQ_1, for all the projects described in Table 2, the set of features reported in Table 1 are computed. Therefore, a TCN classifier is built to predict the presence of software defects on the analyzed systems. The performance of the TCN variant is evaluated to answer RQ_1. Moreover, to answer RQ_2, the performance of TCN is also compared to the performances obtained by using alternative networks like LSTM and CNN-2D. Referring to RQ_3, an additional TCN classifier is built. It uses only product metrics as features. Finally, to investigate the RQ_4, we calculated the distribution of clean and buggy revisions for each project as reported in Table 2.

In all the classifications, a hyper-parameters optimization [11] is performed (all the possible parameters' combinations are evaluated, considering later only the best) to obtain the suitable performance of the considered classifiers.

The architecture parameters evaluated in the hyper-parameter optimization step as reported in Table 3 are the network size, the activation function, the learning rate, the number of layers, the batch size, the optimization algorithm, and sliding window size (for TCN and LSTM classifiers). The selection of the best parameters is performed with an SBMO (Sequential Bayesian Model-based Optimization) approach implemented through the use of the TPE (Tree Parzen Estimator) algorithm [13, 14]. Looking at Table 3, the ranges evaluated for all the considered parameters are reported:

– **Network size**. Three groups of network sizes are considered: small, medium, and large. The small size network includes many learning parameters lower or equal to 1.5 million. For the medium size, the included number of parameters is between 1.5 million and 7 million. Finally, for the large, the included number of parameters is between 7 million up to 12 million parameters.

- **Activation function**. Three activation functions are evaluated: Relu, Swish and Mish [58].
- **Learning rate**. The evaluated learning rate is between 5 to 15, normalized on the base of the used optimization algorithm. For example, when the optimization algorithm is SGD, it is between 0.005 and 0.015.
- **Number of layers**: The considered number of layers is between 5 and 9.
- **Batch size**. The considered batch size is between 16 and 512.
- **Optimization algorithm**. The more used optimization algorithm is the stochastic gradient descent (SGD) [59]. Other evaluated algorithms are: Adam [43], RmsProp [67], Nadam [67], Adamax [64] and Adagrad [64]. Notice that, coherently to [63], the optimization algorithms are configured (when it was possible) by using the accelerated gradient correction Nesterov (NAG). This allows to (*i*) avoid excessive and unnecessary changes in the parameter space, and (*ii*) improve learning performance.
- **Dropout rate**: The dropout rate ranges between 0.10 and 0.25.

The neural network training is then performed by using as a loss function the cross-entropy [45].

Moreover, to improve the classification performance, great attention is paid at avoiding over-fitting. However, the regularization is ensured through an early stopping: when the validation loss of the trained model remains constant for ten consecutive epochs, the training is stopped. Successively, the best-obtained model is persisted, stored, and used in the testing.

The network architecture is implemented by using Tensorflow[3] and Keras[4].

Tensorflow is an open-source software library for high-performance numerical calculations. Keras is a high-level neural networks API based on Python (the network is developed with Python language).

Finally, the accuracy of the classifiers is evaluated by using the $F1$ score defined as the harmonic average of the precision (p) and the recall (r). In particular, p is the ratio between the number of corrected positive results and the number of all positive results returned by the classifier and r is the ratio between the number of corrected positive results and the number of all relevant results.

[3] https://www.tensorflow.org.

[4] https://keras.io.

6 Discussion of results

In the following sub-sections, the obtained results for each research question described in Sect. 5.1 are reported. The last sub-section also reports a possible application scenario of the proposed approach.

6.1 RQ_1: TCN performance

The best F1 measures for the proposed TCN variant are reported in Table 6. The table reports, for each evaluated project, the permutation providing the best F1 value. Looking at this table, we observe that the best F1 is obtained when networks with six and seven layers are used. The F1 is between 0.9462 (for Commons Imaging) and 0.9615 (for ZooKeeper) while the best optimization algorithm is always SGD. As a remark, we state that the obtained F1 scores are in almost all cases and for all analyzed projects greater than the best values known in the literature. However, in Table 4 we report the F-measure obtained by existing alternative approaches. In the first column of the table, the name of the approaches and their reference are reported. In the second column, for each considered approach, the language of the analyzed projects is displayed while the third column described the adopted prediction model. Finally, in the last column, the F-measure range for the approach is reported. Referring to the F-measure, notice that in all the cases our obtained values are higher if compared with existing approaches. The only exception is represented by the approach proposed in [44]. This approach, differently from ours, uses high-level data extracted from the PROMISE dataset and does not exploits commit-level historical data. As a consequence, in a real scenario, this approach allows performing defect prediction only across releases resulting less flexible and useful. This consideration can be also extended to [27] where the authors have conducted the study on ten real-world datasets from NASA projects. However, the extracted high-level data is not usable to evaluated the approach capability to predict defects at the commit level.

Another consideration regards the adopted metrics. The approaches in Table 4 never include process metrics that should be useful to monitor the activities performed in the frame of each commit also taking into account the context in which they occur. Moreover, in all the cases no quality metrics are used. This limits the capability to predict if a given change induces a fix since there isn't any information about the quality of both the changed source code elements and the activities performed around each commit.

Table 4 Comparison with alternative approaches

Reference	Language	Prediction model	F-measure range
Intra-projects defect prediction			
Deeper [70]	C, Java	DBN	0.22–0.63
Manjula et al. [44]	C, C++	DNN	0.79–0.98
Kamei et al. [41]	C, C++, Java	Logistic Regression	0.10–0.60
TLEL [69]	C, C++, Java	Decision Tree, Random Forest	0.10–0.68
DSL [72]	C, C++, Java	Logistic Regression, k-nearest neighbors, Random Forest, Extremely Randomized Trees, XGBoost	0.25–0.67
Pascarella et al [52]	C, C++, Java, JavaScript, Ruby, Perl	Random Forest	0.60–0.73
PHIForest [27]	C, C++	Random Forest-based	0.50–0.87
Hoang et al. [35]	C++, Python	CNN	Not available
Cross-projects defect prediction			
Kamei et al. [40]	C, C++, Java, JavaScript, Ruby, Perl	Random Forest	0.18–0.70
Cong [39]	C, C++, Java, JavaScript, Ruby, Perl	DA-KTSVM	0.18–0.63

6.2 RQ_2: LSTM and CNN-2D performance

Tables 7 and 8 respectively report the best F1 scores obtained by all LSTM and CNN-2D permutations. Even if both the LSTM and the CNN-2D give good results, the accuracy and the F1 values obtained by the TCN are always greater for all the systems. There is only one exception represented by the Jackson DF project where the values of F1 and accuracy for LSTM and TCN are almost equivalent but for the LSTM a larger network is required (seven layers instead of six) leading to slightly worse training times (eight seconds per epoch instead of six seconds, in our experimental environment). The worst performances are always obtained by the CNN-2D classifier. Finally, in the last column of tables 6, 7 and 8 we report the average training times (seconds per epoch). These values show that the training times for TCN, LSTM, and CN-2d are quite comparable. Generally, TCN requires lower time and CNN-2D higher time. Consider that the times are strongly conditioned by the adopted parameters, for example, the very low value obtained for Jackson Core using LSTM (Table 7) is due to the batch size fixed to 64. Similarly, the higher value obtained in Table 6 for JFreeChart can be motivated by the fact that the batch size is fixed to 16. To better investigate this aspect, Fig. 4 shows the boxplots of times obtained for TCN (green box plots) and LSTM (yellow box plots) when the same combinations of the number of layers and batch size are used. Observing this figure some useful considerations can be made. First of all, it shows that batch size value strongly influences the times: greater is the batch size lower are the times, in particular when the batch size is fixed to 64 the times are the lowest ones. Secondly, it highlights how the box plots are similar and, as consequence, also the times they represent. Concerning the overall data spread only in one case the box plots significantly differ, for configuration 6–64, where the LSTM presents more scattered data than TCN.

It should be noted that also in the case of LSTM and CNN-2D the obtained results are better in all cases and for all projects than the best ones known in the literature. In our view, these good results could depend not only on the networks used but also on how we built the moving window over the complete time series as well as the features used.

6.3 RQ_3: Product and process metrics evaluation

Our research aims at predicting the software defectiveness at a class level granularity across software project commits,

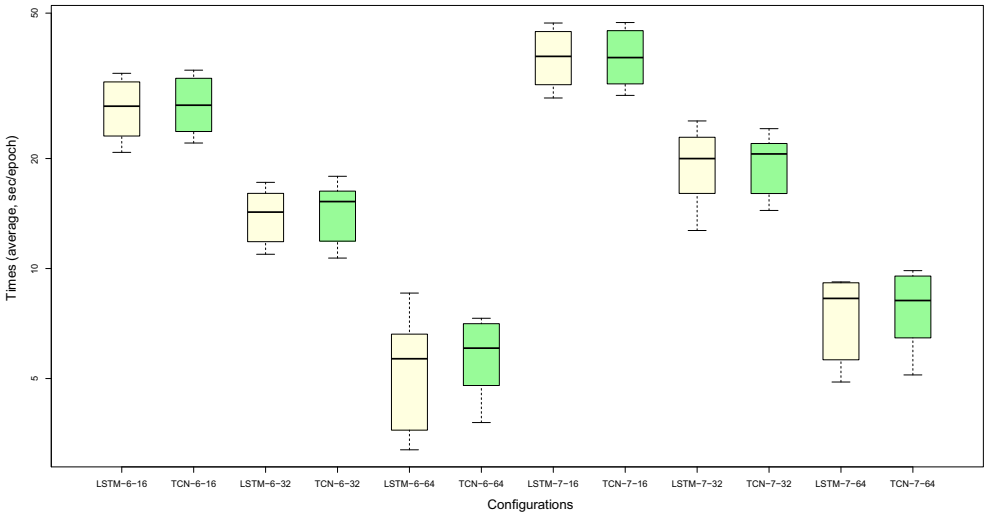

Fig. 4 Box plots of the times for different TCN and LSTM configurations by number of layers and the batch size (the x labels are in the format `<network-kind>-<number-of-layers>-<batch-size>`)

using both process and product metrics. In the literature, about the combined use of both types of metrics, there are controversial results. Some investigations, experimented with both product and process metrics for bug prediction, finding that product metrics are poorer [30]. Other investigations, instead, stated that process features can complement the capabilities of product predictors for bug prediction [49, 50]. Finally, an extensive comparison of bug prediction approaches relying on both product and process metrics, finding that no technique works better in all contexts [28].

For this reason, in the context of RQ_3, we build another class-level bug prediction model relying on only product metrics to investigate if these metrics provide or not a significant contribution when used with process metrics. In Table 9 the best F1 score, obtained by using TCN and only product metrics as features, is reported. Comparing the results reported in Table 9 with those reported in Table 6, we observe that, first of all, the only use of product metrics provide in itself good results and, secondly, that the combined use of product and process metrics in the feature model ensures both higher accuracy and F1.

6.4 RQ_4: Impact of data imbalance

To face the problem of data imbalance there are several techniques already exploited (for example Correlation-based Feature Selection [32] and Random Over-Sampling algorithm [21]). In our work, we did not use any technique to face this problem but, differently, we investigated if the data imbalance affects or not the results in a significant way. For this reason, we calculated the distribution of clean and buggy revisions for each project as reported in Table 5. This table shows that all the datasets are imbalanced. The most imbalanced dataset, JFreeChart, contains only 0.78% defects, while the most balanced dataset, Commons Imaging, contains 22.04% defects. Figure 5, instead, shows the box-plots of the F-measure values for each permutation generated using TCN, where the systems are shown on the X-axis, in ascending order of the imbalance, and the F-measure values on the Y-axis. The box plots, except in the case of JFC, are all comparatively shorts. This suggests that overall permutations have a low degree of dispersion of the F-measure scores. The distributions of F-measure values are symmetric in the case of JC, ZK, CI, while a bit asymmetrical are the distributions in the case of X2J and JDF. The low degree of dispersion of the F-measure suggests that hyper-parameter optimization converges quickly and that, as consequence, is not necessary to run many permutations to achieve a good F result using the TCN. Only in the case of JFC, the highest imbalanced project (below 1% of buggy revision ratio), there is a significant dispersion and skewness of the F-measure values through the permutations with the interquartile range wider than the others, and the worst F1 value less than 0.9. As a consequence, for the JFC system, the above considerations are not valid. Finally, observing Tables 5 and 6, we also note

Table 5 Distributions of clean and buggy revisions for each project

Project	Label	%Clean revisions	%Buggy revisions
JFreeChart	JFC	99.21	0.78
Xerces2 Java	X2J	97.83	2.16
Jackson Core	JC	95.51	4.48
Jackson DF	JDF	92.34	7.65
ZooKeeper	ZK	91.68	8.31
Commons Imaging	CI	77.95	22.04

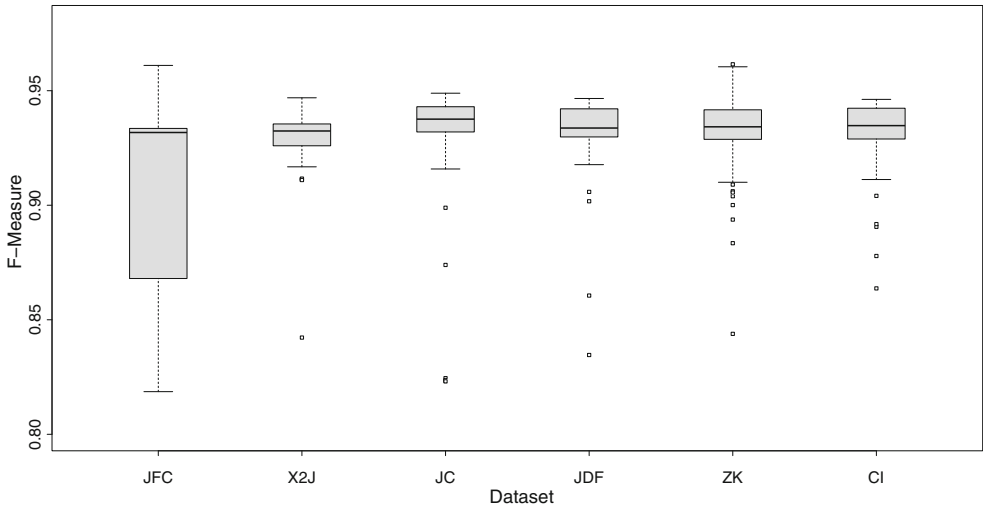

Fig. 5 Box plots of the F-measure for each model configuration using TCN

that the imbalance affects more the process of optimization of the hyper-parameters than the performance.

All the best F-measure values, in fact, are very close, thickening in the range [0.9462, 0.9615] and suggesting that the proposed TCN variant, even when the hyper-parameters optimization process is more unstable and less predictable due to the class imbalance, exhibits a robust behavior.

6.5 Application scenario

Finally, we assessed of the applicability of the proposed model in a real-world scenario, usually characterized by missing or limited historical data (we assume that it is the typical situation of a new project) [73]. In this scenario, the proposed approach can be successfully applied by using cross-project defect prediction (CPDP) ([53]), consisting of transferring prediction models from one or more projects to another. According to this, we applied our approach in a CPDP context training a classifier on all the systems of our study excluding one that is used as a test (i.e., the ZooKeeper project). This means that in this scenario we suppose to perform our prediction on a new project (ZooKeeper) by using historical data extracted from other existing projects (all other projects used in the previous experiment).

The obtained results show an Accuracy of 0.951 and an F-measure of 0.9173. This result is satisfying since in the cross-project defect prediction literature [56, 73], a predictive model is generally evaluated as having good performance when the accuracy is greater than 0.75. However, according to the survey proposed in [42], the existing approaches never achieve 0.75 for F1-score and accuracy simultaneously. Table 4 also reports, in the last two rows, two existing CPDP approaches. The F-measure range shows that in the best case the obtained F-measure is ≈0.7.

Table 6 Permutations providing the best F1 for each project using TCN

Project	Activation function	Learning rate	No. Layers	Batch size	Optimization Algorithm	Dropout rate	Accuracy	Loss	F1	Average Training Time per epoch (sec)
ZooKeeper	mish	3	6	16	SGD	0.2	0.9613	0.1060	0.9615	35
Xerces2 Java	relu	3	6	32	SGD	0.1	0.9463	0.1825	0.9469	6
JFreeChart	mish	6	7	16	SGD	0.15	0.9610	0.0966	0.9610	47
Jackson DF	relu	6	6	32	SGD	0.2	0.9461	0.1878	0.9466	6
Jackson Core	relu	6	7	16	SGD	0.1	0.9508	0.1433	0.9489	22
Commons Imaging	relu	3	7	32	SGD	0.1	0.9449	0.1947	0.9462	11

Table 7 Permutations providing the best F1 for each project using LSTM

Project	Activation Function	Learning Rate	No. Layers	Batch size	Optimization Algorithm	Dropout Rate	Accuracy	Loss	F1	Average Training Time per epoch (sec)
ZooKeeper	mish	3	6	16	SGD	0.2	0.9441	0.1955	0.9443	38
Xerces2 Java	relu	6	7	32	SGD	0.1	0.9446	0.2096	0.9449	8
JFreeChart	relu	6	7	32	SGD	0.2	0.9449	0.1827	0.9456	10
Jackson DF	relu	6	7	32	SGD	0.1	0.9463	0.1655	0.9468	8
Jackson Core	relu	3	6	64	SGD	0.2	0.9447	0.2050	0.9450	2
Commons Imaging	relu	3	7	16	SGD	0.1	0.9449	0.1896	0.9453	15

Table 8 Permutations providing the best F1 for each project using CNN-2D

Project	Activation function	Learning rate	No. layers	Batch size	Optimization algorithm	Dropout rate	Accuracy	Loss	F1	Average training time per epoch (sec)
ZooKeeper	relu	6	7	16	SGD	0.15	0.8884	0.4876	0.8852	69
Xerces2 Java	relu	3	6	64	Nadam	0.15	0.9259	0.2753	0.9261	15
JFreeChart	relu	3	6	64	Nadam	0.15	0.9125	0.3252	0.9453	21
Jackson DF	relu	3	6	64	Nadam	0.15	0.8793	0.4961	0.8872	20
Jackson Core	swish	3	6	64	Nadam	0.15	0.9450	0.1479	0.9450	24
Commons Imaging	relu	6	6	64	SGD	0.2	0.8033	0.7746	0.8137	18

Table 9 Permutations providing the best F1 for each project using TCN and only product metrics

Project	Activation function	Learning rate	No. layers	Batch size	Optimization algorithm	Dropout rate	Accuracy	Loss	F1
ZooKeeper	relu	3	7	32	SGD	0.15	0.8636	0.3518	0.9063
Xerces2 Java	relu	3	7	32	SGD	0.15	0.9138	0.2853	0.9056
JFreeChart	relu	3	7	32	SGD	0.15	0.8640	0.7337	0.8642
Jackson DF	relu	3	7	32	SGD	0.15	0.9166	0.4769	0.9166
Jackson Core	relu	3	7	32	SGD	0.15	0.9239	0.3251	0.9268
Commons Imaging	relu	3	7	32	SGD	0.15	0.9072	0.5826	0.9453

7 Threats to validity

In this section, the threats to the validity of the investigation are discussed.

Construct Validity: A threat to construct validity concerns the reliability of the source code measurement tool our study is based on. Since we are not able to establish the degree to which a tool yields consistent results we decided to use three different tools [3, 34, 60]. In this way, we are able to check whenever possible and necessary, if the measures obtained from one tool are different from those calculated by the other ones. Moreover, all the three tools used are publicly available to make it possible to replicate the measurement task in other studies.

Internal Validity: The threat to internal validity concerns factors that can influence our observations. Particularly, whether the metrics are meaningful to our conclusions and

whether the measurements are adequate. To this aim, an accurate process for the data gathering has been performed.

External Validity: the threat to external validity concerns the generalization of our results. Although we investigated six well-known OSS systems different for dimensions, domain, size, timeframe, and the number of commits, we are aware that a further empirical validation on commercial systems would be beneficial to better support our findings. Commercial systems differ from OSS systems for the nature of reported defects. In commercial systems, the defects are reported only by customers for released versions while in OSS systems the defects can be reported by developers during development activities and by customers, for stable releases. Another limit of our work is that we considered only systems written in Java because the tools used work only on Java programs. Thus, we cannot claim generalization concerning systems written in different languages as well as to projects belonging to industrial environments.

8 Conclusions and future work

The proposed study describes an approach based on deep learning for just-in-time defect prediction. It specifically considers a variant of temporal convolutional networks to predict changes that will introduce software defects. The features model used is based on a set of fine-grained quality metrics. For the empirical experimentation, a data-set has been obtained collecting the data from six open-source software projects, specifically through the assessment of 36 class level source code metrics and 7 process metrics, all detected commit by commit. Results for the performed evaluation highlight that the predictions obtained using the proposed approach are satisfying, indeed, the F-measure obtained is always greater than the 0.94, achieving the 0.96 value in the case of the ZooKeeper and JFreeChart projects. To our knowledge, it is the best result in related literature for the JIT technique. Moreover, our approach provides an interesting result of data-set imbalance. It always gives good results with a data-set whose proportion of minority class is at least 1%. In these cases, the TCN converges quickly to the best F1 result. Only when the degree of imbalance is extreme, i.e., the proportion of the minority class is lower than 1% of the data-set, the TCN does not quickly converge to the best result.

The main limitation of the proposed model is related to the large amount of historical data required to train it to perform well. However, this limitation is common to all predictive approaches. In practice, training data may not be available for projects in the initial development phases, or for legacy systems where historical data are often not stored. For this reason, as future work, we aim to validate the applicability of the proposed model in a cross-project context, that is, a model trained using historical data from a project and tested on other projects. The preliminary evaluation carried out in this work, performed on just one model, seems to be promising and deserving further studies. Of course, we intend to extend the set of metrics considered as features also including process metrics. Finally, we plan to conduct a controlled study with practitioners to evaluate the efficacy of our model in-field. This could allow to make defect prediction more usable in practice and support in real-time development activities, for example during code review activities and/or code writing.

Declaration

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Ackerman LBA, Lewski F (1989) Software inspections: an effective verification process. IEEE Softw 6:31–36. https://doi.org/10.1109/52.28121
2. Ahmad J, Farman H, Jan Z (2019) Deep learning methods and applications, pp 31–42. Springer Singapore. https://doi.org/10.1007/978-981-13-3459-7_3
3. Aniche M (2015) Java code metrics calculator (CK). Available in https://github.com/mauricioaniche/ck/
4. Ardimento P, Aversano L, Bernardi ML, Cimitile M (2020) Temporal convolutional networks for just-in-time software defect prediction. In M. van Sinderen, H. Fill, L.A. Maciaszek (eds.) Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020, pp 384–393. ScitePress. https://doi.org/10.5220/0009890003840393
5. Ardimento P, Aversano L, Bernardi ML, Cimitile M, Iammarino M (2021) Temporal convolutional networks for just-in-time design smells prediction using fine-grained software metrics. Neurocomput 463:454–471. 10.1016/j.neucom.2021.08.010. https://www.sciencedirect.com/science/article/pii/S0925231221011942
6. Ardimento P, Bernardi ML, Cimitile M (2018) A multi-source machine learning approach to predict defect prone components. In Proceedings of the 13th International Conference on Software Technologies, ICSOFT 2018, Porto, Portugal, July 26-28, 2018, pp 306–313. https://doi.org/10.5220/0006857803060313
7. Aversano L, Bernardi ML, Cimitile M, Iammarino M, Romanyuk K (2020) Investigating on the relationships between design smells removals and refactorings. In M. van Sinderen, H. Fill, L.A. Maciaszek (eds.) Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020, pp 212–219. ScitePress. https://doi.org/10.5220/0009887102120219
8. Bai S, Kolter JZ, Koltun V (2018) An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. CoRR arXiv:1803.01271

9. Barnett JG, Gathuru CK, Soldano LS, McIntosh S (2016) The relationship between commit message detail and defect proneness in java projects on github. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016, pp 496–499. https://doi.org/10.1145/2901739.2903496
10. Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Software Eng 22(10):751–761. https://doi.org/10.1109/32.544352
11. Bengio Y (2000) Gradient-based optimization of hyperparameters. Neural Comput 12(8):1889–1900. https://doi.org/10.1162/089976600300015187
12. Bengio Y, Courville A, Vincent P (2014) Representation learning: a review and new perspectives
13. Bergstra J, Bardenet R, Bengio Y, Kégl B (2011) Algorithms for hyper-parameter optimization. In Proceedings of the 24th International Conference on Neural Information Processing Systems, NIPS'11, p 2546–2554. Curran Associates Inc., Red Hook, NY, USA
14. Bergstra J, Yamins D, Cox DD (2013) Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML'13, p I–115–I–123. JMLR.org
15. Bernardi M, Cimitile M, Martinelli F, Mercaldo F (2018) Driver and path detection through time-series classification. J Adv Transp 2018. https://doi.org/10.1155/2018/1758731
16. Bernardi ML, Cimitile M, Martinelli F, Mercaldo F (2019) Keystroke analysis for user identification using deep neural networks. In 2019 International Joint Conference on Neural Networks (IJCNN), pp 1–8. https://doi.org/10.1109/IJCNN.2019.8852068
17. Bird C, Nagappan N, Murphy B, Gall H, Devanbu PT (2011) Don't touch my code!: examining the effects of ownership on software quality. In: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011, pp 4–14. ACM
18. Boucher A, Badri M (2016) Using software metrics thresholds to predict fault-prone classes in object-oriented software. In 2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science Engineering (ACIT-CSII-BCD), pp 169–176. https://doi.org/10.1109/ACIT-CSII-BCD.2016.042
19. Brito e Abreu F, Melo W (1996) Evaluating the impact of object-oriented design on software quality. In: Proceedings of the 3rd International Software Metrics Symposium, pp 90–99. https://doi.org/10.1109/METRIC.1996.492446
20. Cabral GG, Minku LL, Shihab E, Mujahid S (2019) Class imbalance evolution and verification latency in just-in-time software defect prediction. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp 666–676. https://doi.org/10.1109/ICSE.2019.00076
21. Chawla NV (2009) Data mining for imbalanced datasets: an overview. In Data mining and knowledge discovery handbook, pp 875–886. Springer
22. Chen X, Zhao Y, Wang Q, Yuan Z (2018) Multi: Multi-objective effort-aware just-in-time software defect prediction. Inf Softw Technol 93:1–13
23. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Software Eng 20(6):476–493. https://doi.org/10.1109/32.295895
24. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493. https://doi.org/10.1109/32.295895
25. Dabic O, Aghajani E, Bavota G (2021) Sampling projects in github for MSR studies. In: Proceedings of the 18th International Conference on Mining Software Repositories, MSR'21, p. To appear. arXiv:2103.04682
26. Dam HK, Tran T, Pham TTM, Ng SW, Grundy J, Ghose A (2018) Automatic feature learning for predicting vulnerable software components. IEEE Trans Softw Eng
27. Ding Z, Xing L (2020) Improved software defect prediction using pruned histogram-based isolation forest. Reliab Eng Syst Saf 204:107170
28. DAmbros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir Softw Eng 17(4–5):531–577
29. Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In 19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands, p 23. IEEE Computer Society
30. Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661
31. Greiler, M., Herzig, K., Czerwonka, J. (2015) Code ownership and software quality: a replication study. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp 2–12. 10.1109/MSR.2015.8
32. Hall MA (1999) Correlation-based feature selection for machine learning. Ph.D. thesis, Department of Computer Science, University of Waikato, The address of the publisher
33. Hassan AE (2009) Predicting faults using the complexity of code changes. In 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, pp 78–88. https://doi.org/10.1109/ICSE.2009.5070510
34. Hilton R J: Java Source Metrics (2009 (accessed January 16, 2020)). Available in https://github.com/rodhilton/jasome
35. Hoang T, Dam HK, Kamei Y, Lo D, Ubayashi N (2019) Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp 34–45. IEEE
36. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
37. Ioffe S, Szegedy C (2015) Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15, pp 448–456. JMLR.org. http://dl.acm.org/citation.cfm?id=3045118.3045167
38. Jahanshahi H, Jothimani D, Başar A, Cevik M (2019) Does chronology matter in jit defect prediction? a partial replication study. In Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, pp 90–99
39. Jin C (2021) Cross-project software defect prediction based on domain adaptation learning and optimization. Expert Syst Appl 171(114637):1. 10.1016/j.eswa.2021.114637. https://www.sciencedirect.com/science/article/pii/S0957417421000786
40. Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE (2016) Studying just-in-time defect prediction using cross-project models. Empir Softw Eng 21(5):2072–2106. https://doi.org/10.1007/s10664-015-9400-x
41. Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time

quality assurance. IEEE Trans Softw Eng 39(6):757–773. https://doi.org/10.1109/TSE.2012.70
42. Khatri Y, Singh SK (2021) Cross project defect prediction: a comprehensive survey with its swot analysis. Innovations in Systems and Software Engineering pp 1–19
43. Kingma DP, Ba J (2014) Adam: A method for stochastic optimization
44. Manjula C, Florence L (2019) Deep neural network based hybrid approach for software defect prediction using software metrics. Clust Comput 22(4):9847–9863. https://doi.org/10.1007/s10586-018-1696-z
45. Mannor S, Peleg D, Rubinstein R (2005) The cross entropy method for classification. In Proceedings of the 22Nd International Conference on Machine Learning, ICML '05, pp 561–568. ACM, New York, NY, USA
46. Misra D M: A self regularized non-monotonic neural activation function (arXiv pre-print, 2019)
47. Mitchell TM (1997) Machine learning, 1st edn. McGraw-Hill Inc, New York, NY, USA
48. Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Techn J 5(2):169–180
49. Moser R, Pedrycz W, Succi G (2008) Analysis of the reliability of a subset of change metrics for defect prediction. In Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, pp 309–311
50. Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, pp 181–190. https://doi.org/10.1145/1368088.1368114
51. Myers GJ, Sandler C (2004) The art of software testing. Wiley, Hoboken, NJ, USA
52. Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. J Syst Softw 150:22–36. https://doi.org/10.1016/j.jss.2018.12.001
53. Peters F, Menzies T, Marcus A (2013) Better cross company defect prediction. In 2013 10th Working Conference on Mining Software Repositories (MSR), pp 409–418. IEEE
54. Phan AV, Nguyen ML, Bui LT (2018) Convolutional neural networks over control flow graphs for software defect prediction. CoRR arXiv:1802.04986
55. Pickerill P, Jungen HJ, Ochodek M, Mackowiak M, Staron M (2020) PHANTOM: curating github for engineered software projects using time-series clustering. Empir Softw Eng 25(4):2897–2929. https://doi.org/10.1007/s10664-020-09825-8
56. Porto FR, Simao A (2016) Feature subset selection and instance filtering for cross-project defect prediction-classification and ranking. CLEI Electron J 19(3):4
57. Rahman F, Devanbu P (2013) How, and why, process metrics are better. In 2013 35th International Conference on Software Engineering (ICSE), pp 432–441. IEEE
58. Ramachandran P, Zoph B, Le QV (2017) Searching for activation functions. CoRR arXiv:1710.05941
59. Schaul T, Antonoglou I, Silver D (2013) Unit tests for stochastic optimization
60. Spinellis D (2005) Tool writing: a forgotten art? (software tools). IEEE Softw 22(4):9–11. https://doi.org/10.1109/MS.2005.111
61. Staudemeyer RC, Rothstein Morris E (2019) Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks. arXiv e-prints arXiv:1909.09586
62. Subramanyam R, Krishnan M (2003) Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Trans Softw Eng 29:297–310. https://doi.org/10.1109/TSE.2003.1191795
63. Sutskever I, Martens J, Dahl G, Hinton G (2013) On the importance of initialization and momentum in deep learning. In Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML'13, pp III–1139–III–1147. JMLR.org
64. Vani S, Rao TVM (2019) An experimental approach towards the performance assessment of various optimizers on convolutional neural network. In 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), pp 331–336. https://doi.org/10.1109/ICOEI.2019.8862686
65. Varma S, Simon R (2006) Bias in error estimation when using cross-validation for model selection. BMC Bioinf 7:91. https://doi.org/10.1186/1471-2105-7-91
66. Wang T, Zhang Z, Jing X, Zhang L (2016) Multiple kernel ensemble learning for software defect prediction. Autom Softw Eng 23(4):569–590. https://doi.org/10.1007/s10515-015-0179-1
67. Wang Y, Liu J, Mišić J, Mišić VB, Lv S, Chang X (2019) Assessing optimizer impact on DNN model sensitivity to adversarial examples. IEEE Access 7:152766–152776. https://doi.org/10.1109/ACCESS.2019.2948658
68. Xu Z, Li S, Xu J, Liu J, Luo X, Zhang Y, Zhang T, Keung J, Tang Y (2019) LDFR: learning deep feature representation for software defect prediction. J Syst Softw. https://doi.org/10.1016/j.jss.2019.110402
69. Yang X, Lo D, Xia X, Sun J (2017) Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. Inf Softw Technol 87:206–220
70. Yang X, Lo D, Xia X, Zhang Y, Sun J (2015) Deep learning for just-in-time defect prediction. In 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015, pp 17–26. https://doi.org/10.1109/QRS.2015.14
71. Yang Z, Yang D, Dyer C, He X, Smola A, Hovy E (2016) Hierarchical attention networks for document classification. In Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp 1480–1489. Association for Computational Linguistics, San Diego, California. https://doi.org/10.18653/v1/N16-1174. https://www.aclweb.org/anthology/N16-1174
72. Young S, Abdou T, Bener A (2018) A replication study: just-in-time defect prediction with ensemble learning. In Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, pp 42–47
73. Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp 91–100