This paper has been accepted by QRS 2018 (the IEEE 18th International Conference on Software Quality, Reliability and Security).

Cross-Entropy: A New Metric for Software Defect Prediction

Xian Zhang, Kerong Ben, Jie Zeng
Department of Computer and Data Engineering
Naval University of Engineering
Wuhan, China
tomtomzx@foxmail.com, benkerong08@163.com, zyj183247166@qq.com

***Abstract*—Defect prediction is an active topic in software quality assurance, which can help developers find potential bugs and make better use of resources. To improve prediction performance, this paper introduces *cross-entropy*, one common measure for natural language, as a new code metric into defect prediction tasks and proposes a framework called *DefectLearner* for this process. We first build a recurrent neural network language model to learn regularities in source code from software repository. Based on the trained model, the cross-entropy of each component can be calculated. To evaluate the discrimination for defect-proneness, cross-entropy is compared with 20 widely used metrics on 12 open-source projects. The experimental results show that cross-entropy metric is more discriminative than 50% of the traditional metrics. Besides, we combine cross-entropy with traditional metric suites together for accurate defect prediction. With cross-entropy added, the performance of prediction models is improved by an average of 9.0% in F1-score.**

Keywords—software defect prediction; natural language processing; language model; code naturalness; deep learning

I. INTRODUCTION

The accurate prediction of where defects are likely to occur in code is important since it can help direct test effort, reduce costs and improve the quality of software [30]. As a consequence, defect prediction attracts a lot of attention [1][3][6][21][24], and has become an important area of research in the realm of software quality assurance.

Typically, defect prediction modelling is a process of training a learner from a corpus of software data (e.g., static code metrics, change information) and applying the model to identify the buggy-proneness of new data. The performance of prediction is mainly influenced by two aspects: a modeling technique and software metrics. To build an effective model for defect recognition, many machine-learning algorithms have been adopted [24], including logistic regression, naive Bayes, ensemble learning [12], dictionary learning [37], transfer learning [16], and deep neural network (DNN) [9][38].

Meanwhile, many researchers devoted their efforts to design new metrics (also called *features*) for defect prediction [8][29], e.g., Halstead metrics [20] based on operator and operand counts, McCabe metrics [33] based on dependencies, CK metrics [27] based on method and inheritance counts, etc., QMOOD metrics [13] based on measure of cohesion, aggregation and data access, etc., code change metrics [32] including number of lines of code added, removed, etc., and other process metrics [17]. In general, there are two common points among these metrics: all designed manually by human experts and often fail to capture the semantic information of programs. To bridge this gap, Wang et al. [29] first proposed to leverage a deep learning model, deep belief network (DBN), to learn semantic representation of programs from source code. As their evaluation results illustrate, the semantic features significantly improve defect prediction compared to traditional metrics. However, the features generated by DNNs are lack of interpretability, i.e., no real meanings, thereby hardly being utilized to instruct engineers in programming. Moreover, when the automatically learned features are combined with traditional hand-engineered metrics [15], more promotion can be produced, which means the existence of the complementarity between these two types of metric.

Overall, there is no one metric set can perform well in any case yet. It indicates that the aforementioned metrics are inadequate representation and abstraction of software defects, relative to the diversity of defect patterns. For this reason, this paper attempts to absorb different elements to characterize defects.

In recent years, with the wide application of modern software, the large and growing software repositories have gathered tremendous amount of source code and other meta-data. This abundance of artifacts motivates new, data-driven approaches to analyze software [18]. From the perspective of natural language processing (NLP), a series of research has been conducted to study the statistical properties of code [2][4][18][28][42]. Hindle et al. [2] first presented that programs are mostly simple and repetitive, and thus they have usefully predictable statistical properties that can be captured in language models and leveraged for software engineering tasks, which is called the ***naturalness hypothesis***. Learning from a corpus, a language model will assign low *cross-entropy* (or high probability) to code that appears often in practice, i.e., is natural (cf. Section II). Further, based on this work, Ray et al. [4] suggested another hypothesis: *unnatural code is more likely to be buggy*, and first presented evidence that code with bugs tends to be more entropic, becoming less so as bugs are fixed. Their experiments also show that language models work as well as popular tools like FindBugs in static bug detection. Wang et al. [28] presented a similar work, in which they also

leveraged *n-gram* language models to detect bugs that got better results than rule-based approaches. Even though, these methods have not seen much industrial uptake. One possible cause is their imprecision [18], which leads to a lot of manual verification.

Inspired by the work above, we plan to absorb cross-entropy as a new code metric into file-level defect prediction to complement the existing metric sets. In fact, Ray et al. [4] were the first to use (only) cross-entropy scores for defect prediction. However, the advantage of the entropy-based predictor is limited and the evaluation is brief to some extent. To bridge this gap, we will present a comprehensive evaluation for the effectiveness of cross-entropy metric.

This paper develops a framework called *DefectLearner* to automatically measure the cross-entropy value of each software component and complete the subsequent defect prediction tasks. To make use of the advantages of deep structure in feature learning, we build a long short-term memory (LSTM) cell based recurrent neural network (RNN) for programming language modeling. Through mining a corpus of code tokens, the model can well capture the common patterns and regularities in programs. Then, the trained model is utilized to estimate the joint probability distributions of input token sequences, and thus the cross-entropy scores of software components can be derived. Finally, combining cross-entropy with traditional metrics and the label data, a typical classify-cation workflow is established for defect-proneness prediction.

To assess the validity of the new metric, we design the experiments from two perspectives. Focusing on a corpus of 12 Java open-source projects, we first measure the class-discrimination of the entropy metric by a comparison with 20 widely used traditional metrics. The results show that cross-entropy is more discriminative than half of baseline metrics in three relevance criteria. The other method is to check how much improvement of performance the cross-entropy can make on different datasets. In the experiments, four common classifiers and performance measures are adopted. As the empirical results demonstrate, when this new feature is integrated into traditional metric suites, the scores of *Precision*, *Recall*, *F1*, and *AUC* measure get an absolute increase of 4.8%, 1.8%, 2.8%, and 1.8% respectively on average. In conclusion, the cross-entropy metric plays a positive role in different defect-proneness prediction tasks.

In summary, the main contributions of this paper are:

- From the point of view of code naturalness, we introduce cross-entropy as a new software metric into typical defect prediction at file level.
- We give a comprehensive evaluation of the effectiveness of cross-entropy metric. The experimental results suggest that cross-entropy is complementary to traditional metrics in defect prediction tasks.
- We propose a RNN-based defect prediction framework, called *DefectLearner*, to automatically generate the cross-entropy feature of software components, and combine other metrics together for the identification of buggy code.

The rest of this paper is organized as follows. Section II introduces the background of our work. Section III describes the overall framework of our proposed approach and elaborates the techniques we applied. Section IV presents the experiments and results. Section V discusses the related work. Finally, we summarize and present our conclusion and future work in the last section.

II. BACKGROUND

In this section, we give a brief introduction to the background of language models and cross-entropy measure.

A. Language model

Statistical language models are a kind of basic model in NLP [2][7], aiming to learn the joint probability function of sequences of words in a language [39]. More formally, consider a set of allowable lexical units Ω, and the set of all possible sequences Ω^*. For this paper, lexical unit refers to code token. Given a token sequence $S = w_1 w_2 \cdots w_l$ ($w_i \in \Omega$, $S \in \Omega^*$), the probability of this sequence occurring can be estimated as a product of a series of conditional probabilities for each token's occurrence:

$$\begin{aligned} P(S) &= P(w_1)P(w_2 \mid w_1)\cdots P(w_l \mid w_1, w_2, \cdots, w_{l-1}) \\ &= \prod_{i=1}^{l} P(w_i \mid w_1, \cdots, w_{i-1}) \end{aligned} \quad (1)$$

where l denotes sequence length. In practice, given a corpus C of programs ($C \subseteq \Omega^*$), we can use a language model to estimate the parametric probability $P(S, \boldsymbol{\theta})$ in the maximum-likelihood way, where $S \in C$ and $\boldsymbol{\theta}$ is the parameters to be estimated. To simplify the calculation of $P(w_i \mid w_1, \cdots, w_{i-1})$, the classic *n-gram* model is proposed, with a *Markov assumption*: the conditional probability of a token is dependent only on the n-1 most recent tokens. Thus (1) is approximated as:

$$P(S) = \prod_{i=1}^{l} P(w_i \mid w_1, \cdots, w_{i-1}) \approx \prod_{i=1}^{l} P(w_i \mid w_{i-(n-1)}, \cdots, w_{i-1}) \quad (2)$$

For (2), the simple maximum-likelihood based frequency-counting of token sequences is usually adopted to estimate the conditional probabilities:

$$P(w_i \mid w_{i-(n-1)}, \cdots, w_{i-1}) = \frac{count(w_{i-(n-1)}, \cdots, w_{i-1}, w_i)}{count(w_{i-(n-1)}, \cdots, w_{i-1})} \quad (3)$$

Although *n-gram* models have been simplified in terms of probability calculation, it is still difficult to learn long distance information. This is mainly because the models regard language units as isolated symbols, which cannot effectively use the semantic relations between tokens, thereby causing the curse of dimensionality. For example, given a 10-gram model with a vocabulary of size 2,000, there are about 10^{33} different probabilities to be estimated, which will lead to memory explosion and data sparseness in the frequency-counting of (3).

Neural language models are a class of language model designed to overcome the curse of dimensionality problem for modeling language sequences by using the semantic similarity between words. Commonly, the distributed representation of

words and neural network technique are used together to estimate the joint probability distribution of sequences. The neural language model was formally proposed by Bengio et al. in 2001 [39], and has made great progress in many NLP tasks [11][14][40], such as machine translation, information retrieval, and dialogue systems.

Neural language models, on one hand, utilize word embedding method to map discrete language symbols into a set of dense vectors. In this continuous space, similar words are likely to have similar vectors, which can enhance the learning of semantic relations between words. On the other hand, leveraging highly nonlinear neural networks to fit the occurrence probabilities of sequences, the model can describe a language more flexibly. Particularly, with the rise of deep learning (e.g., RNNs), the capability of learning long-term dependencies has been significantly promoted. From the analysis above, we can see that neural language models have prominent advantages in language modeling. Therefore, in this paper, we will take advantage of DNN for programming language modeling.

B. Cross-Entropy

In information theory, cross-entropy is commonly used to quantify the difference between two probability distributions [2][7]. In this paper, the cross-entropy has two roles. One is as a performance criterion for evaluating language model. The other is as a code metric denoting the naturalness (i.e., the likelihood of occurrence) of software components.

In a corpus, common code relatively own high probability of appearance, namely low information entropy. In this case, a (trained) language model will assign low estimated entropy to these code. Here, the estimated entropy is the so-called cross-entropy, whose infimum is the true entropy. Thus, in order to let the model get better estimation, we need the whole cross-entropy on the corpus to be smaller. For this reason, cross-entropy (or its exponential form, *perplexity*) is often adopted for language model validation [2][7]. After the validation, we can use the trained model to measure the naturalness of software components for defect prediction.

Formally, given a corpus (or a component, etc.) $C=\{S_1 S_2 \cdots S_N\}$ ($C \subseteq \Omega^*$) and a language model M , the cross-entropy of the model on the corpus can be calculated by:

$$\begin{aligned} CE_M(C) &= -\frac{1}{l_C}\log_2 P_M(C) \\ &= -\frac{1}{l_C}\sum_{j=1}^{N}\log_2 P_M(S_j) \\ &= -\frac{1}{l_C}\sum_{j=1}^{N}\sum_{i=1}^{l_j}\log_2 P_M\left(w_{j,i}|w_{j,1},\cdots,w_{j,i-1}\right) \end{aligned} \tag{4}$$

where $S_j \in C$ is the j-th sequence in the corpus, N is the total number of sequences, l_j is the length of S_j, $l_C=\sum_{j=1}^{N} l_j$ is the total length of the sequences in the corpus, $P_M(S_j)$ is the probability of occurrence of S_j estimated by M , $P_M(C)=\prod_{j=1}^{N} P(S_j)$ is the probability of occurrence of the corpus.

Perplexity is an exponential form of cross-entropy, which is defined as follows:

$$PP_M(C)=2^{CE_M(C)} \tag{5}$$

where $CE_M(C)\in[0,+\infty)$ and $PP_M(C)\in[1,+\infty)$. In fact, perplexity represents the reciprocal of the geometric mean of conditional probabilities $P_M\left(w_{j,i}|w_{j,1},\cdots,w_{j,i-1}\right)$, i.e., the reciprocal of the average probability of each token guessed successfully by the language mode.

III. Approach

In this section, we present the details of our proposed approach *DefectLearner*.

A. Overall Framework

Fig. 1 illustrates the overview of our proposed *DefectLearner*, a framework designed to automatically generate cross-entropy scores of software components with arbitrary granularity, and combine traditional features together for accurate software defect prediction. As the figure shows, the framework mainly contains three phases: a learning phase (Phase 1), a measuring phase (Phase 2), and a prediction phase (Phase 3).

In the learning phase, our goal is to build a neural language model to learn the common patterns or usage regularities of a programming language by mining software repository (Section III-B). We first carry out data preprocessing on the collected projects, such as removing comments and tokenization, etc. Next, using word-embedding method, the token sequence can be represented as a set of real valued vectors. Then we build a multi-layer LSTM network to learn the implied sematic features from the input token vectors.

After training and testing, the language model, which can define an occurrence probability over each input sequence of tokens, will be utilized to measure the cross-entropy metric of software components of interest. This is the main task of the measuring phase (cf. Section III-C). In this process, what gets measured actually is the source code of components. Thus, we need extract the code areas from repository, and do the same data preprocessing in Phase 1.

Finally, in the prediction phase, we introduce the cross-entropy metric into typical defect prediction problem (cf. Section III-D). We can employ different measurements to extract different features of the evaluated components, or just use the off-the-shelf metric-set data if available. Besides, each component is assigned a “buggy” or “clean” label depend on whether its code region contains defects. Afterwards, we combine the cross-entropy metric with other features together to feed classifiers for the prediction of defect-proneness.

What should be noted is that, the defect information (i.e., the label data of components) is not used to build language model, which means the language-learning phase can be seen as an unsupervised process in terms of defect prediction. Moreover, the training samples applied in Phase 1 can be different from the data in Phase 2 and Phase 3, but the investigated components should be matched with each other in the last two phases.

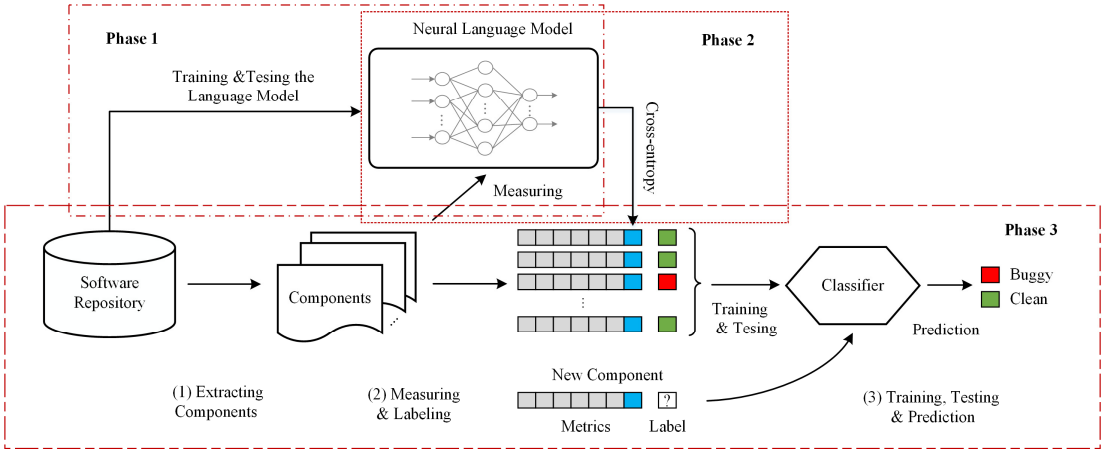


Fig. 1. The overall workflow of *DefectLearner*

B. *Neural language model*

This paper presents a LSTM-based recurrent neural network language model for programming language processing. Fig. 2 contains the details.

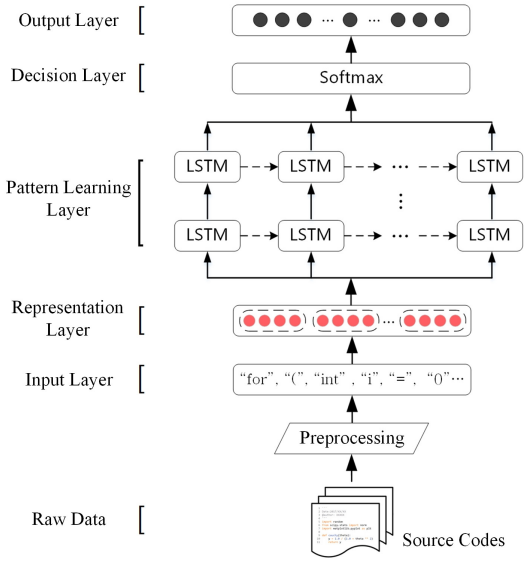


Fig. 2. The neural language model designed in *DefectLearner*

In the language modeling, we first collect software projects of interest from archives and create a corpus. To analyze the input raw code fragments, we need to conduct essential data preprocessing to generate token sequences for language learning. Next, the sequential samples are feed to the core part of the model, a DNN, which is used to automatically learn contextual information and hidden features. Finally, a multi-class classifier is connected in the end to predict the next token of each input sequence, and the obtained probability distribution can be used to define the likelihood of occurrence of the sequence.

The main components of the neural language model are described in detail below.

1) Data preprocessing:

Before training the neural network, data preprocessing is executed. The specific steps are introduced as follows:

Step1: Generate effective code. Remove the comments and blank lines, retaining effective lines of code.

Step2: Tokenization. According to language specification, the code files are lexically analyzed to produce token sequences, for example, " 'for', '(', 'int', 'i', '=', '0', ';' … ".

Step3: Create a vocabulary. Merge source code files, count unique tokens and their frequencies, and build a vocabulary.

Step4: Replace rare tokens. To control the complexity of the model, we need to remove low frequency tokens. In this paper, the total number of tokens is set to V , which means that Top V high frequency tokens are reserved, and the rest are replaced by "<RARE>". We also replace *String* by "<STSING>", since too many natural language words contain.

Step5: Build training set and testing set for model validation. In this step, long token fragments are cut into a series of sequences with equal length for feeding model.

2) Network structure:

As shown in Fig. 2, the DNN part of our model mainly includes an input layer, a data representation layer, a pattern-learning layer, a decision layer, and an output layer. The representation layer leverages word-embedding technique to map each code token into a high dimensional real-valued vector, in which tokens with similar contexts will be assigned similar vectors. The pattern-learning layer uses highly nonlinear mapping to generate the semantic features of these vector sequences. Next, the decision layer synthesizes this information to predict the next token of the input code line, with an estimated probability distribution output. As the total prediction error drops, we can consider that the language model

has been learning the usage patterns or writing customs of a given language.

The core architecture of our network is a multi-layer LSTM-cell based RNN, which is utilized to extract and integrate dependency relationship from the token sequences. LSTM cell, as one of the most popular and efficient methods for understanding sequential dependencies, has been wildly adopted in building RNNs [11][26][40]. This approach uses memory gates to control its inputs and outputs, which can effectively reduce the effects of vanishing and exploding gradient and enhance the long-term learning ability. The LSTM cell we selected can be formalized as [35]:

$$\begin{aligned} \boldsymbol{i}_t &= \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i\right) \\ \boldsymbol{f}_t &= \sigma\left(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f\right) \\ \boldsymbol{o}_t &= \sigma\left(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o\right) \\ \boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tanh\left(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c\right) \\ \boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{c}_t\right) \end{aligned} \tag{6}$$

where $\sigma(x) = 1/\left(1+e^{-x}\right)$ and $\tanh(x) = \left(e^{x}-e^{-x}\right)/\left(e^{x}+e^{-x}\right)$ are activation functions; $\boldsymbol{W}$ and $\boldsymbol{b}$ are weight matrix and bias vector respectively; $\odot$ denotes element-wise multiplication (i.e., Hadamard product); $\boldsymbol{i}_t, \boldsymbol{f}_t, \boldsymbol{o}_t$ are input gate, forget gate, and output gate; $\boldsymbol{x}_t, \boldsymbol{c}_t, \boldsymbol{h}_t$ are input vector, cell states, and hidden states respectively.

3) Training and testing:

Deep network architectures often encounter many troubles during the training process, such as overfitting, gradient explosion, or gradient vanishing. Hence, this paper adopts several optimization strategies as follows:

- Dropout [22][35]. The key idea of this strategy is to randomly drop units from the neural network during training, which is a very effective regularization technique to prevent over-fitting.
- Gradient clipping [25]. This strategy is simple and effective to prevent gradient explosion by limiting the norm of gradient.
- Adaptive learning rate. The gradient descent algorithm is used to optimize the network, and the learning rate l_{rate} is adjusted dynamically with the number of training epochs:

$$l_{\text{rate}} = \begin{cases} 1, & i < \lambda \\ \eta^{i-\lambda}, & \lambda \le i \le N_{\text{epoch}} \end{cases} \tag{7}$$

  where $i, N_{\text{epoch}} \in \mathbf{Z}^{+}$ denote the i-th epoch and the maximum number of epochs, $\lambda \in [1, N_{\text{epoch}}]$ and $\eta \in (0,1)$ are the parameters of learning rate.

For model testing, we choose 10-fold cross validation with perplexity measure (see (5)) as loss function.

C. Measurement of Software Component

After the learning phase, we use the trained language model to measure the naturalness of software components by (4), i.e., to generate the cross-entropy values for defect prediction. Specifically, we first match the evaluated components with their source code areas in the repository. Before feeding the code to language model, we should do the same preprocessing like Phase 1, in which each component will be cut into fixed-length token sequences. Then, we utilize the trained model to measure these inputs separately. After the measurement, the average cross-entropy of these sequences is the (estimated) naturalness of the software component.

D. Performing Defect prediction

Generally, defect prediction modelling is to build a learner from a corpus of software data (e.g., static code metrics, change information) and apply the model to new data, where the objective of prediction can be set to estimate the number of defects remaining in software systems, or recognize the defect-proneness of software components. The latter is what we are concerned.

The Phase 3 in Fig. 1 depicts a typical process of defect-proneness prediction, which is commonly adopted in previous studies [1][12][15][29][37]. Firstly, we extract components of interest from software repository. Here, the components can be packages, files, or methods, etc. Secondly, we collect a set of metrics of each component by means of different measurements, such as complexity metrics, coupling or cohesion metrics, etc. [8]. Meanwhile, the corresponding category attribute (i.e., buggy or clean) can be labeled depending on the number of defects in each component. Finally, based on machine-learning techniques, a classic framework of binary classification problem can be build. Then we can feed the components with features and labels to classifiers for training and testing, and feed new components without labels to the trained models for prediction.

In this paper, we select four common classifiers for prediction [5], including support vector machine (SVM), logistic regression (LR), naive Bayes (NB), and random forest (RF), which are widely adopted in previous defect prediction research [1][12][15][29][30].

IV. Evaluation

In this section, we will evaluate the effectiveness of our *DefectLearner* framework. All the experiments were performed on a machine with an Intel i7-6700K 4.0 GHz CPU, 16G RAM, and a GeForce GTX 1070 8G GPU. The implementation of the deep learning model and the classifiers were based on *TensorFlow*[1] 1.0.0 and *Scikit-learn*[2] 0.19.1 respectively. The tokenization of Java source code was accomplished via an open-source package called *javalang*[3], whose implementation is based on the Java 8 language specification.

The core task of the evaluation is to verify the validity of the cross-entropy metric. For this purpose, we design two evaluation methods. One is to compare the ability of cross-entropy with other baseline metrics for distinguishing defective

[1] https://www.tensorflow.org/
[2] http://scikit-learn.org/stable/
[3] https://github.com/c2nes/javalang

components. The other is to measure how much contribution the cross-entropy metric can bring to defect prediction.

Based on these methods, our experiments are designed to address the following research questions (RQ):

- RQ1: *Is cross-entropy more class-discriminative than traditional code metrics in defect prediction?*
- RQ2: *Can cross-entropy improve the performance of prediction models?*
- RQ3: *How is the performance of cross-entropy on different datasets?*

A. Dataset Description

To make it easier to repeat and verify our experiments, we use publicly available data from the tera-PROMISE Repository[4]. We select 12 Java open-source projects from this repository, whose basic information is shown in TABLE I. These datasets have been widely used in defect prediction studies [1][12][15][16][29]. As illustrated, the numbers of evaluated components of the projects range from 51 to 965, and the buggy rates have a minimum value of 11.36%, a maximum value of 63.58%, and an average of 31.13%. The more detailed description of the datasets can be obtained from [21]. Based on this information, we extracted the whole source code of these projects from Apache[5], SourceForge[6] or GitHub[7] archive correspondingly, and applied these data to our framework *DefectLearner*. Statistically, these projects contain a total of 8,883 source code files with about 1.78 M LOCs.

TABLE I. THE INVESTIGATED PROJECTS

Project	Version	Evaluated Components	Buggy Components	Buggy Rate
Ant	1.7.0	745	166	22.28%
Camel	1.6.0	965	188	19.48%
Ivy	2.0.0	352	40	11.36%
jEdit	4.2	367	48	13.08%
Log4j	1.1	109	37	33.95%
Lucene	2.4.0	340	203	59.71%
PBeans	2.0	51	10	19.61%
POI	3.0	442	281	63.58%
Synapse	1.2	256	86	33.59%
Velocity	1.6.1	229	78	34.06%
Xalan-J	2.6.0	885	411	46.44%
Xerces	1.3.0	453	69	15.23%
Total	—	5,194	1,617	31.13%

Each component in a project represents a Java class file of the release and consists of two parts: instance features including 20 static code metrics and a class label "bug" indicating how many defects in that class. The 20 traditional metrics were collected by Jureczko et al. (i.e., the dataset's donators) [21]. TABLE II lists the brief description of this code metric set that used as the baseline for the following contrast experiments.

[4] http://openscience.us/repo/defect/
[5] http://archive.apache.org/dist/
[6] https://sourceforge.net/projects/
[7] https://github.com/apache/

TABLE II. THE 20 TRADITIONAL METRICS IN PROMISE DATASETS USED AS THE BASELINE

Metric	Description
CK suite (6)	
WMC	Weighted Methods per Class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling Between Object classes
RFC	Response for a Class
LCOM	Lack of Cohesion in Methods
QMOOD suite (5)	
NPM	Number of Public Methods
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Function Abstraction
CAM	Cohesion among Methods of Class
Extended CK suite (4)	
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Method Complexity
LCOM3	Normalized Version of LCOM
Martins metric (2)	
Ca	Afferent couplings
Ce	Efferent couplings
McCabe's Cycloamic Complexity (CC) (2)	
Max (CC)	Maximum CC Values of Methods in the Same Class
Avg (CC)	Mean CC Values of Methods in the Same Class
Other(1)	
LOC	Lines of Code

B. Evaluation Measures

Recall that, to verify the validity of cross-entropy metric, we design two evaluation methods. Next, we will give two kinds of evaluation criteria. Considering different measures have different "preferences", we choose more than one measure to reduce the biases.

1) Discrimination measures.

Discrimination measure addresses how much the relevance is between software metric and the buggy label. Generally, with more class-discriminative metrics, the model will be more likely to get good predicted results. One of the most related research areas is the *feature selection* in machine learning [10], which is a well-established and active field. To contrast different metrics' discrimination power, we select three common filter-based feature selection methods: *Pearson's correlation*, *Fisher's criterion*, and *Information gain*, which have been frequently used in software defect prediction [3][12][43].

a) Pearson's correlation (PC): PC is one of the simplest criteria defined as:

$$PC(X) = \frac{\left|\mathrm{cov}(X,Y)\right|}{\sqrt{\mathrm{var}(X)\cdot\mathrm{var}(Y)}} \tag{8}$$

where *X* and *Y* denote a feature variable and target labels, *cov()* and *var()* denote the covariance and the variance.

b) Fisher's criterion (FC): FC is derived from the Fisher's Linear Discriminant Analysis (LDA) [5]. This statistic is defined to be the ratio of the between-class scatter S_b to the within-class scatter S_w and is given by:

$$FC(X_+, X_-) = \frac{S_b}{S_w} = \frac{\left(\hat{X}_+ - \hat{X}_-\right)^2}{S_+^{\,2} + S_-^{\,2}} \tag{9}$$

where $\hat{X}$ denotes the mean of feature variable X, "+" and "-" denote two classes, and the S is defined as:

$$S^2 = \sum_{x_i \in X} \left(x_i - \hat{X}\right)^2$$

c) Information gain (IG): IG, also known as *Mutual information*, is an entropy-based algorithm [5], which measures the reduction of uncertainty about class label after observing the feature.

$$IG(X,Y) = H(X) - H(X \mid Y) \tag{10}$$

where $H(X)$ and $H(X|Y)$ represent Shannon entropy and conditional entropy respectively ,which defined as:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$

$$H(X|Y) = -\sum_{y \in Y} p(y) \sum_{x \in X} p(x \mid y) \log_2 p(x \mid y)$$

2) Performance measures:

Software defect prediction with the defect-proneness as prediction target can be seen as a typical binary classification problem. The performance of a classifier can be assessed via a confusion matrix. As shown in TABLE III, an instance can be predicted into four possible cases, viz., *true positive* (TP), *false positive* (FP), *true negative* (TN), and *false negative* (FN).

TABLE III. CONFUSION MATRIX

Actual	Prediction	
	Predicted Buggy	Predicted Clean
True Buggy	TP	FN
True Clean	FP	TN

From this matrix, a number of measures can be induced. To evaluate the performance of prediction models with different software metric sets, we choose four widely adopted measures [1][6][12][15][38]: *Precision*, *Recall*, *F1*, and *Area Under Curve (AUC)*. Here is a brief introduction.

a) Precision: Precision measures the ratio of the positive instances returned by a predictor correctly.

$$Precision = \frac{TP}{TP + FP} \tag{11}$$

b) Recall: Recall measures the ratio of the true positive instances actually returned by a predictor.

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

c) F1-score: F1-score combines both precision and recall with a harmonic mean form.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{13}$$

d) AUC: AUC measures the area under receiver operating characteristic (ROC) curve [31], in which the X-axis represents false positive rate (FPR)

$$FPR = \frac{FP}{FP + TN} \tag{14}$$

and the Y-axis represents true positive rate (TPR):

$$TPR = \frac{TP}{TP + FN} \tag{15}$$

Additionally, these four measures above all fall in the range [0, 1] and the higher they are, the better the prediction performance represents.

C. Paramter Settings of DefectLearner

The parameter settings of *DefectLearner* are given empirically as follows: the vocabulary size $V = 5{,}000$; the dimension of word vectors $dim_{\mathrm{w}} = 500$; the number of LSTM layers $N_{\mathrm{layer}} = 2$; the numbers of hidden nodes in each LSTM layer $\boldsymbol{dim}_{\mathrm{h}}$ = [800 1,200]; the keep probability of dropout $p_{\mathrm{keep}} = 0.55$; the number of steps of back-propagation through time, also the length of input sequences, $N_{\mathrm{step}} = 35$; the batch size $N_{\mathrm{batch}} = 30$; the number of epochs $N_{\mathrm{epoch}} = 15$; the maximum gradient norm $grad_{\mathrm{max}} = 5$; the parameters of learning rate λ=11, η=0.5 . In LSTM layers, the initial value of forget gate bias is set to 0.0, and for other parameters the initial values are set randomly according to a uniform distribution of [-0.08, 0.08]. Moreover, the parameters of classifiers (viz., SVM, LR, NB, and RF) are set to default configurations suggested by *Scikit-learn*.

D. Results and Analysis

In *DefectLearner*, the first phase is training language model to learn the probability distributions of the occurrence of code sequences. In the experiments, we extract the whole source code files of the 12 projects from software repository, and then carry out necessary preprocessing on the datasets. Next, we feed well-cut token sequences with fixed length to language model for training. Fig. 3 demonstrates the model's loss curves obtained in 10-fold cross validation, where the average time cost is 8857 seconds.

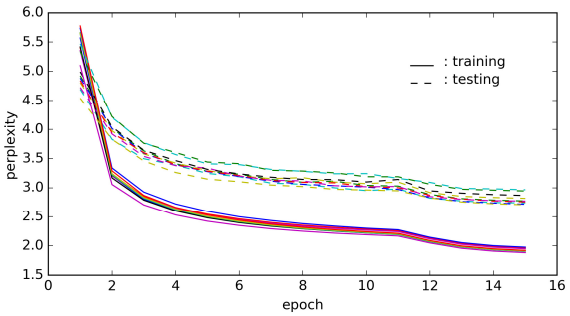


Fig. 3. Loss curves of language model

As shown in Fig. 3, both training curves and testing curves of the ten experiments are very close, which means good model stability. We can also see that the loss curves converge gradually as time goes on. Through 15 epochs of training, the model finally get an average of 1.923 perplexity on training set and 2.806 perplexity on testing set. In fact, the initial perplexity is more than 1,000. It implies that the model does get full training and has learned the common patterns or regularities in

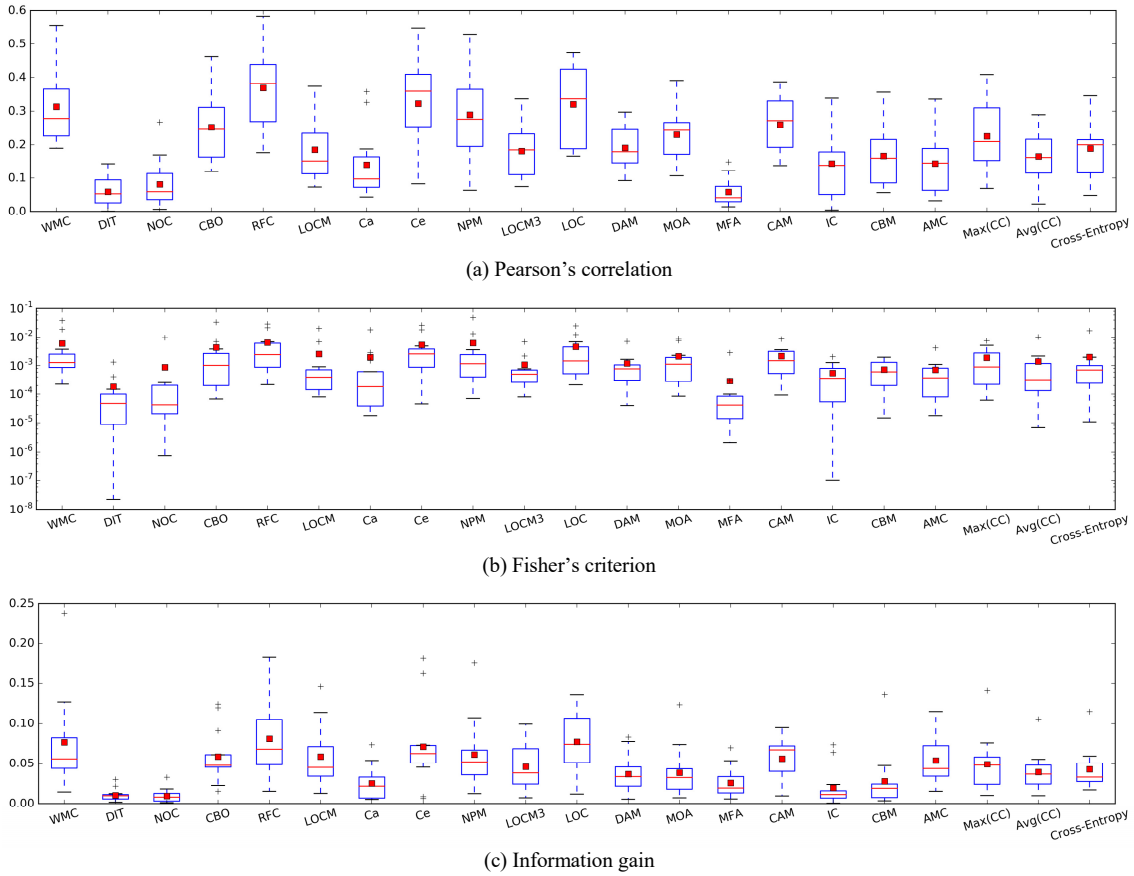
Java code. Besides, the reason why the curves change obviously in the 11th epoch is the decrease of learning rate (see (7)). With this adaptive strategy, the model can be fully optimized by balancing global search and local search.

After the learning phase, we use the trained language model to compute the cross-entropy of code sequences with arbitrary length. As described in Section III-C, we match the evaluated components with corresponding code fragments and execute measurement. Finally, by feeding classifiers the code metric values, defect prediction is conducted.

1) Discrimination of cross-entropy (RQ1).

Generally, the feature more effective in classification tasks tends to own higher ability to distinguish samples. Hence, to assess the significance of cross-entropy, we first measure its discrimination for buggy proneness. As mentioned in Section IV-B, we select three common discrimination measures, viz., *Pearson's correlation*, *Fisher's criterion*, and *Information gain*. Using (8)-(10), we can compute the relevance between cross-entropy metric and the buggy label, namely the discrimination.

Fig. 4 shows the box-plots of the 21 evaluated metrics' discrimination scores on 12 projects, in which the logarithmic coordinate is used in the case of *Fisher's criterion*. From this figure, we can acquire the degree of dispersion and skewness in the data and other key information, such as the median (plotted as red band) and the mean (plotted as red square), etc.



(a) Pearson's correlation

(b) Fisher's criterion

(c) Information gain

Fig. 4. Discrimination scores of the evaluated metrics

As illustrated in Fig. 4, there is a big difference of the 21 metrics' discrimination on the datasets. Taking *Pearson's correlation* as an example, DIT, NOC, Ca, and MFA are obviously worse than other metrics, with almost all values below 0.2, but relatively they own better dispersion (i.e., smaller variance). By contrast, RFC, WMC, and Ce have higher scores, where the median and mean all pass 0.3. However, their values span a very large interval, which means poor stability on different datasets. On 12 projects, the correlation between cross-entropy and buggy labels ranges from 0.048 to 0.345, where the median and mean are 0.197 and 0.189. Compared to other metrics, cross-entropy is at the medium level in terms of both dispersion and mean, which indicates that cross-entropy is more relevant to defect proneness than half of the baseline metrics. What is more, these findings also exist in the cases of *Fisher's criterion*, and *Information gain*.

More explicitly, to give a comparison of discrimination, we sort the 21 metrics by the mean values of measures. TABLE IV lists the ranks of metrics.

TABLE IV. RANKS OF METRICS' CLASS-DISCRIMINATIVE PERFORMANCE

Num	Metric	Pearson	Fisher	Info Gain
1	WMC	4	3	3
2	DIT	20	21	20
3	NOC	19	16	21
4	CBO	7	6	6
5	RFC	1	1	1
6	LOCM	12	7	7
7	Ca	18	11	18
8	Ce	2	4	4
9	NPM	5	2	5
10	LOCM3	13	15	11
11	LOC	3	5	2
12	DAM	10	14	15
13	MOA	8	9	14
14	MFA	21	20	17
15	CAM	6	8	8
16	IC	17	19	19
17	CBM	14	17	16
18	AMC	16	18	9
19	Max(CC)	9	12	10
20	Avg(CC)	15	13	13
21	**Cross-Entropy**	**11**	**10**	**12**

From TABLE IV, we can see three discrimination measures output very similar ranks, where cross-entropy gets 11th, 10th, and 12th respectively and outperforms half of the baseline metrics on average.

Cross-entropy is more class-discriminative than 50% of the traditional code metrics and owns discrimination in defect-proneness prediction.

2) Performance of prediction with cross-entropy (RQ2).

TABLE V. THE PERFORMANCE CHANGES ON DIFFERENT TRADITIONAL METRIC SETS WITH CROSS-ENTROPY ADDED

Measure	Metric Set Name	Metric Set	Metric Set+	Absolute Increase	Relative Increase
Precision	CK	0.5083	0.5133	0.50%	0.98%
	QMOOD	0.5430	0.5671	2.41%	4.45%
	Extended CK	0.4242	0.4636	3.95%	9.30%
	Martins	0.4533	0.5090	5.57%	12.28%
	McCabe	0.5124	0.5412	2.89%	5.63%
	LOC	0.3678	0.5031	13.53%	36.78%
	Average	0.4681	0.5162	**4.81%**	**11.57%**
Recall	CK	0.3248	0.3316	0.68%	2.09%
	QMOOD	0.3532	0.3629	0.98%	2.77%
	Extended CK	0.3069	0.3205	1.36%	4.43%
	Martins	0.2899	0.3106	2.07%	7.15%
	McCabe	0.3548	0.3629	0.81%	2.29%
	LOC	0.2886	0.3398	5.12%	17.76%
	Average	0.3197	0.3381	**1.84%**	**6.08%**
F1	CK	0.3641	0.3674	0.33%	0.91%
	QMOOD	0.3960	0.4123	1.63%	4.11%
	Extended CK	0.3338	0.3468	1.30%	3.91%
	Martins	0.3051	0.3437	3.86%	12.65%
	McCabe	0.3852	0.4036	1.84%	4.76%
	LOC	0.2908	0.3715	8.07%	27.76%
	Average	0.3458	0.3742	**2.84%**	**9.02%**
AUC	CK	0.6813	0.6840	0.27%	0.40%
	QMOOD	0.7149	0.7189	0.40%	0.56%
	Extended CK	0.6533	0.6628	0.95%	1.45%
	Martins	0.6369	0.6610	2.41%	3.78%
	McCabe	0.6775	0.6958	1.83%	2.70%
	LOC	0.6438	0.6940	5.01%	7.78%
	Average	0.6680	0.6861	**1.81%**	**2.78%**

To evaluate a metric, we also want to check how much contribution it can bring. Therefore, we design serval experiments to compare the performance improvement when cross-entropy is mixed with different traditional metric sets. As listed in TABLE II, there are six widely used metric suites, including CK suite, QMOOD suite, Extended CK suite Martins metric, McCabe's CC, and LOC, which characterize different features of software. To give a comprehensive measurement, we combine cross-entropy with the six metric suites together to feed prediction models, where four classifiers (cf. Section III-D) and four performance measures (cf. Section IV-B) are introduced. The experiments are repeated ten times on five-fold cross validation, as illustrated in TABLE V and TABLE VI.

TABLE V presents the performance changes with cross-entropy added, where the results are the mean values of four classifiers on 12 projects, and the "Metric Set+" denotes the new set with the integration of cross-entropy. From this table, we can see cross-entropy does bring improvement in each defect prediction task, no matter using which measures or with which metric sets. Taking F1 measure as an example, cross-entropy contributes an average of 2.84% absolute improvement and 9.02% relative improvement with 6 different suites. The minimum increase is in the case with CK suite, and the maximum increase is in the case with LOC metric. We consider the reason of this result contains two aspects. The first one is that since the six metric suites use different measurement mechanisms, cross-entropy has different degrees of complementarity to them. The other one is different numbers of metrics in these suites. Because the size of defect dataset we used is not large enough, the classifier is more difficult to train with more features input, which influences the contrast results. In conclusion, the results listed in TABLE V explain that cross-entropy is complementary to traditional metric suites.

TABLE VI. THE PERFORMANCE CHANGES OF DIFFERENT CLASSIFIERS WITH CROSS-ENTROPY ADDED

Measure	Classifier	Metric Set	Metric Set +	Absolute Increase	Relative Increase
Precision	SVM	0.4032	0.4303	2.72%	6.74%
	LR	0.5522	0.5718	1.96%	3.55%
	NB	0.4116	0.5273	11.57%	28.11%
	RF	0.5057	0.5355	2.98%	5.89%
	Average	0.4681	0.5162	**4.81%**	**11.07%**
Recall	SVM	0.2744	0.2832	0.88%	3.21%
	LR	0.3223	0.3359	1.35%	4.20%
	NB	0.2800	0.3437	6.37%	22.75%
	RF	0.4020	0.3895	-1.25%	-3.12%
	Average	0.3197	0.3381	**1.84%**	**6.76%**
F1	SVM	0.2882	0.3005	1.23%	4.27%
	LR	0.3603	0.3738	1.35%	3.74%
	NB	0.2947	0.3875	9.27%	31.46%
	RF	0.4401	0.4352	-0.49%	-1.12%
	Average	0.3458	0.3742	**2.84%**	**9.58%**
AUC	SVM	0.6759	0.6927	1.68%	2.49%
	LR	0.7119	0.7112	-0.07%	-0.09%
	NB	0.5862	0.6303	4.41%	7.53%
	RF	0.6979	0.7101	1.22%	1.75%
	Average	0.6680	0.6861	**1.81%**	**2.92%**

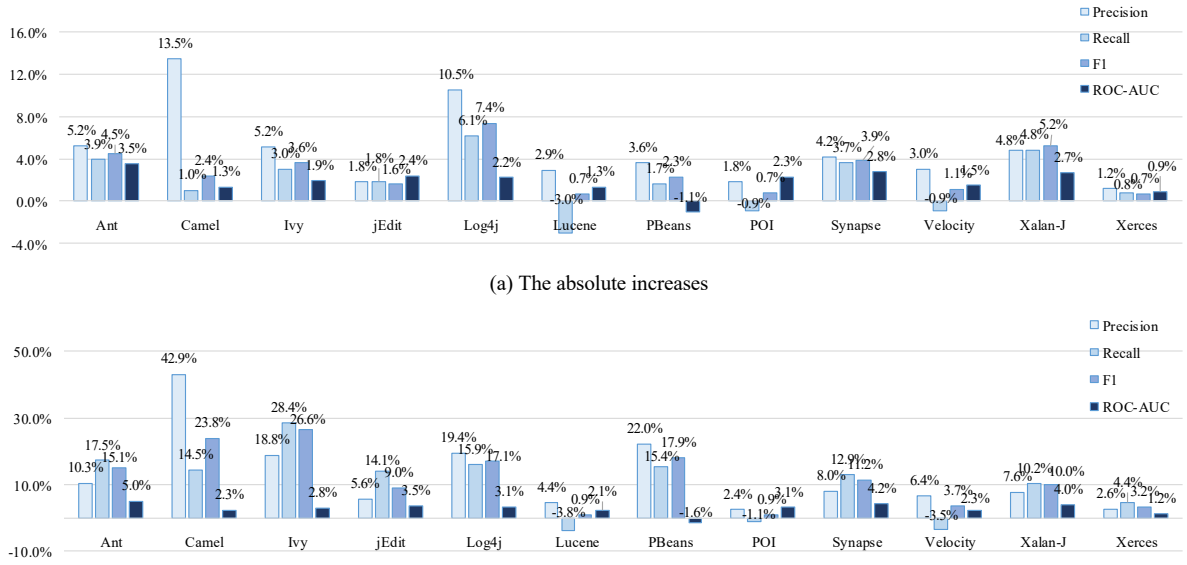
Furthermore, TABLE VI exhibits the performance changes of different classifiers with cross-entropy added, where the results are the mean values of six metric suites on 12 projects.

Besides, to avoid subjectivity, classifiers use the default parameters suggested by *Scikit-learn*. From this table, we can find that four classifiers acquire different performance changes. Roughly, cross-entropy contributes most to NB and least to RF. Even though, cross-entropy does bring improvement in most cases, with an average of absolute increases being 4.81%, 1.84%, 2.84%, and 1.81% respectively in Precision, Recall, F1, and AUC measure.

Cross-entropy is complementary to traditional metric sets in defect-proneness prediction.

3) Performance on different projects (RQ3).

Using similar ways, we next assess the performance of cross-entropy on different datasets. Fig. 5 demonstrates the performance improvement after cross-entropy is integrated, in which the results are the mean values of four classifiers with six metric suites.



(a) The absolute increases

(b) The relative increases

Fig. 5. The performance improvement after cross-entropy metric is added

As Fig. 5 shows, the performance indexes gain a different degree of growth on 12 Java datasets. Roughly, the projects, Camel, Log4j, and Xalan-J, acquire more absolute increases, meanwhile, Lucence, POI, and Velocity are opposite. There is certain degree of change in case of relative comparison, e.g., the growths on Ivy and PBeans are more remarkable. Overall, as we can see from Fig. 5, cross-entropy plays a positive role on the 12 investigated projects, no matter in terms of relative comparison or absolute comparison.

Cross-entropy can improve the prediction performance on different defect datasets.

E. Threats to Validity

There are a number of threats that may affect the results of this study. Relate to internal validity, the data preprocessing method, which is empirical and subjective, may bring about potential threats. As Agrawal et al. [1] suggested, creating better training data is important for learning models, which we will attach more attention to in the future. Another threat is the selection of traditional metrics combined with cross-entropy for defect prediction, which could also produce biases to experiments. To this end, we plan to try more selections and combinations. Threats to external validity concern the generalizability of our research. For project selection, we have tried our best to make our dataset general and representative. However, since the 12 investigated projects are all Java software from tera-PROMISE archive, it is still not generalizable enough to represent all software projects. In the future, we will verify the performance of cross-entropy metric on more open source and private software, and projects written in other languages.

V. Related Work

A. Software Defect Prediction

Software defect prediction has become a popular method for early software inspection, which allows engineers to focus development activities on defect-prone code, thereby improving software quality [30]. Aiming at the characteristics of defect data, Jing et al. [37] designed a supervised cost-sensitive dictionary learning approach to predict defective components. Since defect data commonly including irrelevant and redundant features, and missing samples, Laradji et al. [12] combined ensemble-learning models with feature selection to address these issues. To analyze and compare the various feature selection techniques, Xu et al. [43] conducted a comprehensive empirical study to investigate the impact of 32 methods on the defect prediction performance. Considering

default parameter settings usually bring suboptimization to classifiers, Tantithamthavorn et al. [6] studied the effectiveness of an automated parameter optimization method for defect prediction. Nam et al. [16] focused on heterogeneous defect prediction (i.e., different metrics in different projects), whose work shows that it is possible to quickly transfer lessons learned about defect prediction.

The rise of deep learning sparks a growth of interest in the application of this technique to defect prediction. Instead of using hand-designed defect features, Wang et al. [29] proposed to leverage a representation-learning algorithm, DBN, to learn semantic features from token vectors of programs' ASTs for defect-proneness prediction. Following a similar way, Li et al. [15] built a convolutional neural network for semantic feature generation, and combined traditional hand-designed features together to identify defective files. Aiming at the defect prediction in Android binary executables, Dong et al. [9] trained a multi-layer perceptron to learn both token and semantic features of defects in smali files (decompiled files of apks). For just-in-time defect prediction, Yang et al. [38] introduced DBN as an extractor to integrate the existing change features.

B. Deep Learning for Big Code

Of late years, many (open-source) software repositories have accumulated massive software artifacts, such as source code, changes, and bug-fixes, which attracts a way of research at the intersection of deep learning, programming languages and software engineering (see [18] for a survey). To automatically detect software vulnerabilities, Li et al. [41] proposed a bidirectional LSTM-based detection system, called VulDeePecker, which can achieve few false negative errors. Murali et al. [34] developed a Bayesian framework, which combines a topic model and a RNN model, to learn specifications, and queries the learned models to automatically discover subtle API errors in Android applications. Based on a sequence-to-sequence neural network, Gupta et al. [23] present an end-to-end solution, called DeepFix, which can fix multiple common programming errors in a C program. Related to code search and recommendation, Gu et al. [36] applied a RNN encoder-decoder model to generate API usage sequences for a given natural language query. To promote program synthesis systems, Balog et al. [19] developed a DNNs based framework with a domain specific language specification to induce source code from input-output examples.

VI. Conclusion and Future Work

To improve the performance of defect prediction, a series of software metrics has been proposed. However, relative to the diversity of bug patterns, there is still existing the inadequate representation of software defects. From the perspective of code naturalness, this paper introduces cross-entropy metric into defect prediction tasks to complement the existing metric set.

To make use of the advantages of deep architecture in pattern learning, we construct a RNN based framework *DefectLearner* to automatically calculate the cross-entropy value of each software component and perform defect-proneness recognition. Through mining software repository, the designed RNN language model can work well to capture regularities in source code. Using the trained model, the cross-entropy of code fragment can be measured for defect prediction. Then, we present substantial empirical evidence of the effectiveness of this new metric. As the experimental results show, the cross-entropy metric owns more discrimination power for defect classification than half of 20 common metrics, and does bring improvement to prediction models on different projects. Besides, cross-entropy is a learnable feature that exploits the implicit knowledge in software repositories, and also of good interpretability. In conclusion, these findings suggest that cross-entropy is useful to predict software defects and complementary to existing metric suits.

In the future, we would like to conduct experiments on more projects and metric sets to reduce the threats to validity. We also plan to apply cross-entropy metric to cross-project defect prediction, which is more challenging in application. In addition, it would be promising to extend our approach to other programming languages, such as C/C++ and Python.

Acknowledgment

The authors thank the anonymous reviewers for their feedback that helped us improve this paper. This research was partially supported by National Security Program on Key Basic Research Project of China (No. 613315).

References

[1] A. Agrawal and T. Menzies, "Is 'better data' better than 'better data miners'?: On the benefts of tuning SMOTE for defect prediction," in *ICSE'18*: *40th International Conference on Software Engineering*, 2018. https://doi.org/10.1145/3180155.3180197.

[2] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *ICSE'12: Proc. of the International Conference on Software Engineering*, pp. 837-847, 2012.

[3] B. Ghotra, S. McIntosh, and A. E. Hassan, "A large-scale study of the impact of feature selection techniques on defect classification models," in *MSR'17: IEEE/ACM 14th International Conference on Mining Software Repositories*, pp. 146-157, 2017.

[4] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the naturalness of buggy code," in *ICSE'16: Proc. of the International Conference on Software Engineering*, pp. 428-439, 2016.

[5] C. M. Bishop, Pattern recognition and machine learning, New York: Springer, 2006.

[6] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction model," in *ICSE'16: Proc. of the International Conference on Software Engineering*, pp. 321-332, 2016.

[7] D. Jurafsky and J. H. Martin, Speech and language processing, 2nd ed., Upper Saddle River: Pearson/Prentice Hall, 2009.

[8] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no.8, pp. 1397-1418, 2013.

[9] D. Feng, J. Wang, Q. Li, and S. Zhang, "Defect prediction in Android binary executables using deep neural network," *Wireless Personal Communications*, 2017. https://doi.org/10.1007/s11277-017-5069-3.

[10] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16-28, 2014.

[11] H. Salehinejad, J. Baarbe, S. Sankar, J. Barfett, E. Colak, and S. Valaee, "Recent advances in recurrent neural networks," *arXiv preprint, arXiv*: 1801.01078, 2018.

[12] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features." *Information and Software Technology*, vol. 58, pp. 388-402, 2015.

[13] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4-17, 2002.

[14] J. Hirschberg and C. D. Manning, "Advances in natural language processing," *Science*, vol. 349, no. 6245, pp. 261-266, 2015.

[15] J. Li, P. He, J. Zhu, and R. L. Michael, "Software defect prediction via convolutional neural network," in *QRS'17: Proc. of the International Conference on Software Quality, Reliability and Security*, pp. 318-328, 2017.

[16] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering* , 2017. DOI: 10.1109/TSE.2017.2720603.

[17] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? An empirical study," *Software Quality Journal*, vol.23, no.3, pp. 393-422, 2015.

[18] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of machine learning for big code and naturalness," *arXiv preprint, arXiv*: 1709.06182, 2017.

[19] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to write programs," *in ICLR'17: Proc. of the International Conference on Learning Representations*, 2017.

[20] M. H. Halstead, "Elements of Software Science (Operating and Programming Systems Series)," New York: Elsevier Science Inc., 1977.

[21] M. Jureczko, and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," *In Proc. of the 6th International Conference on Predictive Models in Software Engineering*, 2010.

[22] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R.Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of machine learning research*, vol. 15, no. 1, pp. 1929-1958, 2014.

[23] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: fixing common C language errors by deep learning," in *AAAI'17: Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 1345-1351, 2017.

[24] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504-518, 2015.

[25] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *ICML'13*: *International Conference on Machine Learning*. pp. 1310-1318, 2013.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476-493, 1994.

[28] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tian, "Bugram: Bug detection with n-gram language models," in *ASE'16: Proc. of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 708-719, 2016.

[29] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *ICSE'16: Proc. of the International Conference on Software Engineering*, pp. 297-308, 2016.

[30] T. Hall, S. Beecham, D. Bowes, D. Grayc, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering,". *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276-1304, 2012.

[31] T. Fawcett, "An introduction to ROC analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861-874, 2006.

[32] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," *In ASE'13: Proc. of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 279-289, 2013.

[33] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320, 1976.

[34] V. Murali, S. Chaudhuri, and C. Jermaine. "Bayesian specification learning for finding API usage errors," in *FSE'17: Proc. of the Joint Meeting on Foundations of Software Engineering*, pp. 151-162, 2017.

[35] W. Zaremba, I. Sutskever, and O. Vinyals. "Recurrent neural network regularization," *arXiv preprint, arXiv*: 1409.2329, 2014.

[36] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *FSE'16*: *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 631-642, 2016.

[37] X. Jing, S. Ying, Z. Zhang, S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *ICSE'14: Proc. of the International Conference on Software Engineering*, pp. 414-423, 2016.

[38] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *QRS'15: Proc. of the International Conference on Software Quality, Reliability and Security*, pp. 17-26, 2015.

[39] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," in *NIPS'01: Proc. of Advances in Neural Information Processing Systems*,pp. 932-938, 2001.

[40] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning," *Nature*, vol. 512, no. 7553, pp. 436-444, 2015.

[41] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, and S.Wang, et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *NDSS'18*: *Proc. of the Network and Distributed System Security Symposium*, 2018.

[42] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *FSE'14: Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 269-280, 2014.

[43] Z. Xu, J. Liu, Z. Yang, G. An, and X. Jia, "The impact of feature selection on defect prediction performance: An empirical comparison," in *ISSRE'16: IEEE 27th International Symposium on Software Reliability Engineering*, pp. 309-320, 2016.