This paper has been accepted by ISSSR 2018 (The 4th International Symposium on System and Software Reliability).

Using Cross-Entropy Value of Code for Better 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 meaningful since it can assist software inspection by predicting defective code locations and improve software reliability. Many software features are designed for defect prediction models to identify potential bugs, but no one feature set can perform well in most cases yet. To make better defect prediction, this paper proposes a new code feature, the cross-entropy value of the sequence of code's abstract syntax tree nodes (CE-AST), and develops a neural language model for feature measurement. To evaluate the effectiveness of CE-AST, we first investigate its discrimination for defect-proneness. Experiments on 12 Java projects show that CE-AST is more discriminative than 45% of twenty widely used traditional features. Furthermore, we investigate CE-AST's contribution to defect prediction. Combined with different traditional feature suites to feed prediction models, CE-AST can bring performance improvements of 4.7% in Precision, 2.5% in Recall and 3.5% in F1 by average.**

Keywords—software reliability; defect prediction; natural language processing; language model; code naturalness; cross-entropy

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 [1]. Typically, 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 instances. The latter is this paper concerned.

To improve defect-proneness prediction performance, many software features (also called metrics) are proposed [2][3], e.g., McCabe features [4], CK features [5], QMOOD features [6], code change features [7], and other process features [8]. These above features are often fail to capture the semantic information of programs. To bridge this gap, Wang et al. [9] first proposed to leverage a deep neural network (DNN) model to automatically generate semantic features of programs, which significantly improve defect prediction compared to traditional features. Moreover, when the automatically learned semantic features are combined with traditional hand-designed features [10], more promotion can be produced. Overall, there is no one feature set can perform well in any case yet. It indicates that relative to the diversity of defect patterns, the aforementioned features are the inadequate representation of software defects. For this reason, this paper attempts to characterize defects from different perspective for better defect prediction.

In recent years, based on natural language processing (NLP) methods, a series of research has studied the statistical properties of code [11][12][13][14]. Hindle et al. [11] first presented the **naturalness hypothesis**: "programs that real people actually write 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". 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). In [11][11], the authors used the classic n-gram language model for code naturalness analysis. Based on this work, Ray et al. [12] suggested another hypothesis: unnatural code is more likely to be buggy**.** They first presented evidence that **code with bugs tends to be more entropic**, becoming less so as bugs are fixed. Wang et al. [13] leveraged n-gram language models to detect bugs and got better results than rule-based approaches. Allamanis et al. [14] survey the work of machine learning for code naturalness.

Inspired by the work above, this paper plans to absorb cross-entropy as a new feature for better defect prediction. Considering that the code's abstract syntax tree (AST) contains rich semantic and structural information, we propose to use the cross-entropy value of the sequence of AST nodes (denoted as CE-AST) as the new code feature. To automatically compute CE-AST values of evaluated software instances, we develops a neural language model NLM4Code, whose core part is a long short-term memory (LSTM) cell based recurrent neural network (RNN). Through mining a corpus, the model can well capture the patterns and regularities in programs. Then, the trained model is utilized to estimate the CE-AST scores of software instances. Finally, we combine CE-AST with traditional features for typical defect-proneness prediction (cf. Section III).

To assess the effectiveness of the new feature CE-AST, we design the experiments from two perspectives. On a corpus of 12 Java projects, we first compare the class-discrimination of the entropy feature with 20 widely used traditional features. The results

show that CE-AST is more discriminative than almost half of baseline features. The other perspective is to verify how much improvement of performance that CE-AST can make to defect prediction models. As the empirical results demonstrate, when CE-AST is integrated into traditional feature suites, the scores of Precision, Recall, and F1 get an absolute increase of 4.73%, 2.52%, and 3.54% respectively on average.

II. BACKGROUND

A. Language model

Language model is a kind of basic model in NLP [15], aiming to learn the joint probability function of sequences of words in a language. Given a token sequence $S = w_1 w_2 \cdots w_l$, the probability of this sequence occurring can be estimated as a product of a series of conditional probabilities:

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

where l denotes sequence length. In practice, given a corpus C of programs, we can use a language model to estimate $P(S)$ in the maximum-likelihood way. N-gram models [15] is one class of the most classic language models, but they are difficult to learn long distance information, since they regard language units as isolated symbols, which cannot effectively use the semantic relations between tokens, thereby causing the curse of dimensionality.

Neural language models are 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 [16] and neural network technique are used together to estimate the joint probability distribution of sequences (see (1)). The neural language model was formally proposed by Bengio et al. in 2001 [17], and has made great progress in many NLP tasks [16][18], such as machine translation, information retrieval, and dialogue systems. At present, RNN is the most commonly used form of language modeling [19]. Fig. 1 depicts the scheme of basic RNN language model [20], where hidden states $\boldsymbol{h}_i = \sigma(\boldsymbol{W}_{ih}\boldsymbol{w}_i + \boldsymbol{W}_{hh}\boldsymbol{h}_{i-1})$, outputs $\boldsymbol{y}_i = softmax(\boldsymbol{W}_{ho}\boldsymbol{h}_i)$, $\sigma(x) = 1/(1+e^{-x})$ is sigmoid activation function, the classifier $softmax(x_m) = e^{x_m} / \sum_j e^{x_j}$, and $\boldsymbol{W}$ denotes weight matrix. The input vector $\boldsymbol{w}_i$ denotes the input word encoded using one-hot coding, and the output layer produces a probability distribution over words. The hidden layer maintains a representation of the context, i.e., the sequence history. The input vector and the output vector have dimensionality of the vocabulary. In this framework, the word distributed representations (also called word embeddings or word vector) are found in the columns of $\boldsymbol{W}_{ih}$, with each column representing a word (see [20][21] for details).

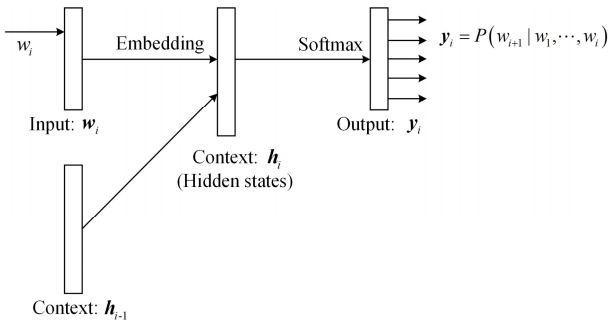

Fig. 1. Scheme of basic RNN language model

B. Cross-Entropy

In information theory, cross-entropy is commonly used to quantify the difference between two probability distributions [15]. In this paper, it is used as a code feature which denotes the naturalness (i.e., the likelihood of occurrence) of software instances [11][12] (see Section I). As Ray et al. [12] present, *unnatural code is more likely to be buggy*, which means that code with bugs tends to be more entropic. Hence, we attempt to use the cross-entropy feature of software instances for defect prediction.

In a corpus, common codes relatively own high probability of appearance, namely low information entropy. In this case, a (trained) language model will assign low estimated entropy to these codes. Oppositely, uncommon or unnatural codes will be measured high estimated entropy. Here, the estimated entropy is the so-called cross-entropy. Formally, given a corpus (or an software instance, etc.) $C = \{S_1 S_2 \cdots S_N\}$ and a language model M, the cross-entropy of the model on the corpus can be calculated as follows:

$$
\begin{aligned}
CE_M(C) &= -\frac{1}{l_C}\log_2 P_M(C) \\
&= -\frac{1}{l_C}\sum_{j=1}^{N}\log_2 P_M(S_j) \\
&= -\frac{1}{l_C}\sum_{j=1}^{N}\sum_{i=1}^{l_j}\log_2 P_M(w_{j,i}|w_{j,1},\cdots,w_{j,i\text{-}1})
\end{aligned}
\tag{2}
$$

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

$$
PP_M(C)=2^{CE_M(C)} \tag{3}
$$

where $CE_M(C)\in[0,+\infty)$ and $PP_M(C)\in[1,+\infty)$. This paper uses perplexity as the loss function for language model training.

III. Defect Prediction with Cross-Entropy Feature

In this section, we present the details of our method for defect prediction with CE-AST as a new feature. We first show the overall framework of our study, and then elaborate the developed language model NLM4Code for measuring codes' cross-entropy.

A. Overall Framework

Fig. 2 illustrates the whole process of defect prediction used in our study, a framework designed to automatically generate cross-entropy scores of software instances and combine traditional features together for accurate defect-proneness prediction.

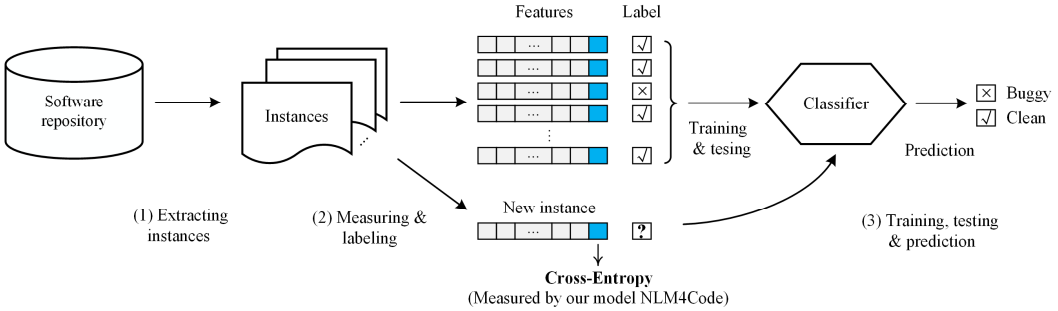

Fig. 2. The process of defect-proneness prediction in our study with CE-AST as a new feature

Firstly, we extract instances of interest from software repository. Here, the instances can be packages, files, or methods, etc. Secondly, we collect a set of features of each instance by means of different measurements, such as complexity features, coupling or cohesion features, etc. [2]. Meanwhile, the corresponding category attribute (i.e., buggy or clean) can be labeled depending on whether its code region contains defects. Finally, based on machine-learning techniques, a classic framework of binary classification problem can be build. Then we can feed the instances with features and labels to classifiers for training and testing, and feed new instances without labels to the trained models for prediction. To make better defect prediction, we propose a new feature CE-AST, i.e., the cross-entropy value of the sequence of AST nodes. For measuring CE-AST, a neural language model NLM4Code is designed, which will be described in detail in Section III-B.

Considering different classifiers have different properties, this paper uses more than one classifier to reduce the biases for evaluation experiments. We select three common classifiers for prediction, including support vector machine (SVM), logistic regression (LR), and naive Bayes (NB), which are widely adopted in previous defect prediction research [1][9][10][22].

B. Language Modeling for CE-AST Calculation

Fig. 3 presents our LSTM-based RNN language model NLM4Code for CE-AST calculation. Before the measurement of cross-entropy feature is model training and testing, and then using the trained model we can compute software instances' CE-AST scores.

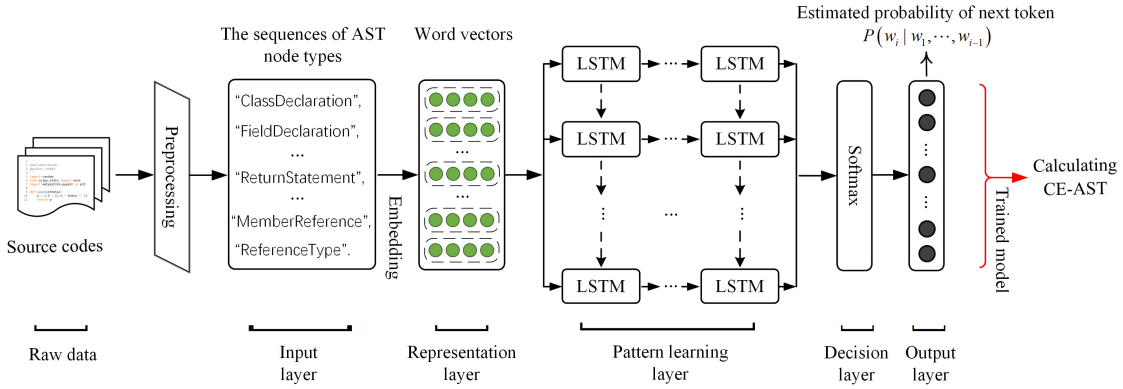

Fig. 3. Our language model NLM4Code used for measuring CE-AST values of software instances

The goal of training NLM4Code is to learn common patterns or regularities in code AST node sequences by mining software repository. We first collect software projects of interest from archives and carry out data preprocessing on the corpus, such as removing comments and parsing, etc. Next, using word-embedding method, the AST node sequence can be represented as a set of real valued vectors. Then we build a multi-layer LSTM network to learn the implied sematic and structural information from the input vectors. Finally, a multi-class classifier (softmax) 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. After training, the language model will be utilized to measure the cross-entropy feature of software instances of interest (see (2)).

The main components of NLM4Code are described in detail below.

1) Data preprocessing

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

Step2: Parse. According to language specification, the source code files are parsed into AST node type sequences, for example, " 'ClassDeclaration', 'FieldDeclaration', 'ReferenceType ', 'VariableDeclarator'…". In this paper, we used an open-source package called *javalang*[1] to parse Java codes, whose implementation is based on the Java 8 language specification.

Step3: Create a vocabulary. Merge AST node type sequence files, count unique tokens, and build a vocabulary with the size denoted as V. TABLE I lists all AST node types used in our study.

Step4: Build a 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.

TABLE I. THE USED AST NODES

AST Node Types				
Annotation	CatchClause	EnumBody	LocalVariableDeclaration	SwitchStatementCase
AnnotationDeclaration	CatchClauseParameter	EnumConstantDeclaration	MemberReference	SynchronizedStatement
AnnotationMethod	ClassCreator	EnumDeclaration	MethodDeclaration	TernaryExpression
ArrayCreator	ClassDeclaration	ExplicitConstructorInvocation	MethodInvocation	This
ArrayInitializer	ClassReference	FieldDeclaration	PackageDeclaration	ThrowStatement
ArraySelector	CompilationUnit	ForControl	ReferenceType	TryStatement
AssertStatement	ConstantDeclaration	FormalParameter	ReturnStatement	TypeArgument
Assignment	ConstructorDeclaration	ForStatement	Statement	TypeParameter
BasicType	ContinueStatement	IfStatement	StatementExpression	VariableDeclaration
BinaryOperation	DoStatement	Import	SuperConstructorInvocation	VariableDeclarator
BlockStatement	ElementArrayValue	InnerClassCreator	SuperMemberReference	WhileStatement
BreakStatement	ElementValuePair	InterfaceDeclaration	SuperMethodInvocation	
Cast	EnhancedForControl	Literal	SwitchStatement	

2) Network structure

As shown in Fig. 3, the DNN part of NLM4Code consists of 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 token into a high dimensional real-valued vector, in which tokens with similar contexts will be assigned similar vectors. The pattern-learning layer

[1] https://github.com/c2nes/javalang

uses highly nonlinear fitting to learn semantic and structural information of these vector sequences. Finally, the decision layer synthesizes this information to predict the next token of the input code line.

The core architecture of our network is a multi-layer LSTM-cell based RNN. LSTM cell [23] is one of the most popular and efficient methods for understanding sequential dependencies [19], which uses memory gates to control its inputs and outputs. The LSTM cell we selected [24] are shown in Fig. 4, where $\sigma(x)=1/(1+e^{-x})$ and $\tanh(x)=(e^{x}-e^{-x})/(e^{x}+e^{-x})$ are activation functions; $\odot$ denotes element-wise multiplication; t denotes recurrent time; $\boldsymbol{i}_t, \boldsymbol{f}_t, \boldsymbol{o}_t$ 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.

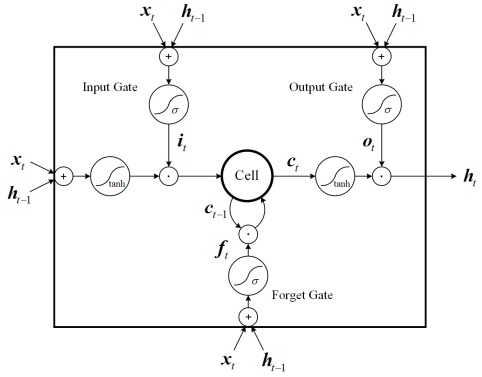

Fig. 4. LSTM cell

3) Model training

For model training, we use gradient descent algorithm to optimize network and choose 10-fold cross validation with perplexity measure (see (3)) as the loss function. To improve model performance, we adopts three network regularization methods: dropout [24], gradient clipping [25], and adaptive learning rate. The learning rate l_{rate} is adjusted dynamically with the number of training epochs:

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

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 the learning rate.

4) Measurement of CE-AST

After the learning phase, we use the trained language model to measure the naturalness of software instances by (2), i.e., to compute the cross-entropy feature values for defect prediction. Specifically, we first extract the evaluated instances' source codes in the repository. Then, we do the same preprocessing in training phase before feeding the code to language model, in which each instance will be parsed into AST and be cut into fixed-length node token sequences. Then, we use the trained model to measure these inputs separately. After that, the average cross-entropy of the sequences is the CE-AST feature score of the instance.

IV. Experiments and Evaluation

In this section, we evaluate the effectiveness of our proposed new code feature CE-AST for defect prediction. For this purpose, we design two kinds of evaluation experiments: 1) to compare the ability of CE-AST with other baseline features for distinguishing defective instances and 2) to measure how much contribution the CE-AST feature can make to defect prediction models.

All experiments were repeated ten times on a computer 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 prediction classifiers were based on TensorFlow[2] 1.0.0 and Scikit-learn[3] 0.19.1 respectively.

A. Datasets

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 II. These datasets have been

[2] https://www.tensorflow.org/
[3] http://scikit-learn.org/stable/
[4] http://openscience.us/repo/defect/

widely used in previous studies [9][10][22][26]. As illustrated, the numbers of evaluated instances 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%. For cross-entropy calculation, we extracted all source codes of these projects from open-source repositories (see [27]), and then fed these data to our designed language model NLM4Code.

TABLE II. THE DATESETS USED IN OUR EXPERIMENTS

Project	Version	Evaluated Instances	Defective Instances	Defective 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/Average	—	5,194	1,617	31.13%

Each instance 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 features were collected by Jureczko et al. [27]. TABLE III lists a brief description of these code features, which will be used as the baseline to evaluate our proposed new feature CE-AST.

TABLE III. THE 20 TRADITIONAL FEATURES USED IN THE DATASETS

Feature Suite	Feature	Description
CK suite	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	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	IC	Inheritance Coupling
	CBM	Coupling Between Methods
	AMC	Average Method Complexity
	LCOM3	Normalized Version of LCOM
Martins suite	Ca	Afferent couplings
	Ce	Efferent couplings
McCabe's Cycloamic Complexity suite	Max (CC)	Maximum CC Values of Methods in the Same Class
	Avg (CC)	Mean CC Values of Methods in the Same Class
Other	LOC	Lines of Code

B. Evaluation Metrics

Considering different metrics have different "preferences", this paper adopts more than one metric to reduce the biases.

1) Discrimination metrics

Discrimination metric addresses how much the relevance is between software feature and the buggy label. Generally, with more class-discriminative features as inputs, classifiers will be more likely to perform well. In this paper, we select two common criteria: Pearson Correlation and Information Gain, which have been frequently used in defect prediction for feature selection [3][26][28].

a) Pearson Correlation (PC): PC is one of the simplest criteria defined as:

$$PC(X) = \frac{\left|\mathrm{cov}(X,Y)\right|}{\sqrt{\mathrm{var}(X)\cdot \mathrm{var}(Y)}} \tag{4}$$

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

b) Information Gain (IG): IG measures the reduction of uncertainty about class label after observing a feature [29].

$$IG(X,Y) = H(X) - H(X \mid Y) \tag{5}$$

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 metrics

Software defect-proneness prediction can be seen as a typical binary classification problem. The performance of classifiers can be assessed via a confusion matrix, as shown in TABLE IV.

TABLE IV. CONFUSION MATRIX

	Predicted Buggy	**Predicted Clean**
Actual Buggy	True Positive (TP)	False Negative (FN)
Actual Clean	False Positive (FP)	True Negative (TN)

From this matrix, a number of metrics can be induced. We choose three widely used metrics [10][22][30]: Precision, Recall, and F1. These metrics all fall in the range [0, 1] and the higher they are, the better the prediction performance represents.

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

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

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

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

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

$$F1 = \frac{2\times Precision\times Recall}{Precision + Recall} \tag{8}$$

C. Model Paramters

The parameter settings of NLM4Code are given as follows: the vocabulary size $V = 63$; the dimension of word vectors $dim_{\mathrm{w}} = 200$; the number of LSTM layers $N_{\mathrm{layer}} = 2$; the numbers of hidden nodes in each LSTM layer $\boldsymbol{dim}_{\mathrm{h}} = [500\ 800]$; the keep probability of dropout $p_{\mathrm{keep}} = 0.55$; the length of input sequences $N_{\mathrm{step}} = 15$; the batch size $N_{\mathrm{batch}} = 30$; the number of epochs $N_{\mathrm{epoch}} = 10$; the maximum gradient norm $grad_{\max} = 4$; the parameters of learning rate λ= 7, 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, to avoid subjectivity, the parameters of classifiers (viz., SVM, LR, and NB) are set to default configurations suggested by Scikit-learn tool.

D. Results and Analysis

In our experiments, the first is training language model NLM4Code. We collect the whole source code files of the 12 investigated projects, and then parse codes into AST node sequences. Next, we feed well-cut token sequences with fixed length to language model. In 10-fold cross validation, NLM4Code get an average perplexity of 2.172 on training set and 2.294 on testing set (see (3)), with average time cost of 1609 seconds. After that, we use the trained language model to compute the cross-entropy of evaluated AST node sequences. Finally, by feeding classifiers code feature set values, defect prediction is conducted.

Recall that, to verify the effectiveness of our proposed new feature CE-AST, we design the evaluation from two perspectives: 1) comparing the discrimination of CE-AST with traditional baseline features for defect-proneness classification and 2) checking how much contribution the CE-AST feature can bring in actual defect prediction tasks.

1) Discrimination of CE-AST

As mentioned in Section IV-B, we select two common discrimination measures, viz., Pearson Correlation and Information Gain. Using (4)-(5), we can compute the relevance between code features and the buggy label, namely the discrimination. TABLE V lists the mean scores of 21 compared features on the datasets described in TABLE II, and their corresponding ranks.

TABLE V. FEATURES' CLASS-DISCRIMINATIVE PERFORMANCE

NO.	Feature	Pearson Correlation		Information Gain	
		Score	Rank	Score ($\times 10^{-2}$)	Rank
1	WMC	0.316	4	7.496	3
2	DIT	0.060	20	1.078	20
3	NOC	0.080	19	0.981	21
4	CBO	0.248	7	5.782	6
5	RFC	0.366	1	8.034	1
6	LOCM	0.193	10	5.736	7
7	Ca	0.131	18	2.398	18
8	Ce	0.321	2	7.132	4
9	NPM	0.289	5	6.055	5
10	LOCM3	0.182	13	4.656	11
11	LOC	0.317	3	7.684	2
12	DAM	0.192	11	3.674	15
13	MOA	0.231	8	3.848	14
14	MFA	0.058	21	2.577	17
15	CAM	0.258	6	5.608	8
16	IC	0.143	17	1.954	19
17	CBM	0.166	14	2.800	16
18	AMC	0.144	16	5.370	9
19	Max(CC)	0.222	9	4.876	10
20	Avg(CC)	0.162	15	3.906	13
21	**CE-AST**	**0.183**	**12**	**4.143**	**12**

As illustrated in TABLE V, there is a big difference of the 21 features' discriminative performance. Taking Pearson Correlation as an example, MFA, DIT, and NOC are obviously worse than other features, with the scores all below 0.100. By contrast, RFC, Ce, and WMC have higher scores that all pass 0.300. Compared to other features, CE-AST is at the medium level. What is more, these findings also exist in the case of Information Gain. From TABLE V, we can also see that the two discrimination metrics output similar ranks, where CE-AST gets both 12th rank and outperforms 45% of the baseline features.

CE-AST is more class-discriminative than 45% of the traditional code features and owns discrimination for defect-proneness identification.

2) Prediction performance with CE-AST

To evaluate a feature, we also want to find out how much contribution it can bring. Therefore, we design experiments to compare the performance improvement when CE-AST is combined with different traditional feature suites. As listed in TABLE III, there are six widely used feature suites, including CK suite, QMOOD suite, Extended CK suite Martins suite, McCabe's CC suite, and LOC, which characterize different aspects of software. To give a comprehensive evaluation, we combine CE-AST with the six feature suites together to feed prediction models respectively to compare the performance changes, where three classifiers (viz., SVM, LR, and NB) and three performance metrics (viz., Precision, Recall, and F1, see(6)-(8)) are used. TABLE VI-VII illustrates the experimental results with five-fold cross validation.

TABLE VI. THE PERFORMANCE IMPROVEMENTS OF DIFFERENT CLASSIFIERS WITH CE-AST FEATURE ADDED

Classifier	Precision				Recall				F1			
	Feature Suite	Feature Suite+	Absolute Increase	Relative Increase	Feature Suite	Feature Suite+	Absolute Increase	Relative Increase	Feature Suite	Feature Suite+	Absolute Increase	Relative Increase
SVM	0.4032	0.4275	2.43%	6.03%	0.2744	0.2797	0.53%	1.95%	0.2882	0.2978	0.96%	3.33%
LR	0.5522	0.5613	0.91%	1.65%	0.3223	0.3307	0.84%	2.60%	0.3603	0.3674	0.71%	1.96%
NB	0.4116	0.5201	10.85%	26.37%	0.2800	0.3419	6.19%	22.10%	0.2947	0.3843	8.95%	30.38%
Average	0.4556	0.5029	**4.73%**	**11.35%**	0.2922	0.3175	**2.52%**	**8.88%**	0.3144	0.3498	**3.54%**	**11.89%**

TABLE VI lists the performance changes of different classifiers with the cross-entropy feature added, where the results are the mean values of 12 projects, and the "Feature Suite+" means the suites with the integration of the new feature CE-AST. To avoid

subjectivity, classifiers use the default parameters suggested by Scikit-learn. From this table, we can find that three classifiers obtain different performance increases, where CE-AST contributes most to NB and least to LR. On average, CE-AST bring absolute improvement of 4.73%, 2.52%, and 3.54% respectively in Precision, Recall, and F1.

TABLE VII. THE PERFORMANCE IMPROVEMENTS ON DIFFERENT PROJECTS WITH CE-AST FEATURE ADDED

Project	Precision				Recall				F1			
	Feature Suite	Feature Suite+	Absolute Increase	Relative Increase	Feature Suite	Feature Suite+	Absolute Increase	Relative Increase	Feature Suite	Feature Suite+	Absolute Increase	Relative Increase
Ant	0.5132	0.5548	4.15%	8.09%	0.1873	0.2315	4.42%	23.61%	0.2582	0.3113	5.31%	20.57%
Camel	0.3229	0.4111	8.82%	27.31%	0.0467	0.0574	1.07%	22.82%	0.0705	0.0894	1.89%	26.87%
Ivy	0.2647	0.3278	6.30%	23.82%	0.0875	0.1278	4.03%	46.03%	0.1153	0.1595	4.42%	38.35%
jEdit	0.2897	0.3316	4.20%	14.49%	0.0990	0.1358	3.68%	37.16%	0.1320	0.1754	4.34%	32.88%
Log4j	0.5209	0.6166	9.57%	18.38%	0.3379	0.3934	5.55%	16.42%	0.3906	0.4581	6.74%	17.27%
Lucene	0.6596	0.6909	3.13%	4.75%	0.8133	0.7833	-3.01%	-3.70%	0.7084	0.7131	0.47%	0.66%
PBeans	0.1596	0.2059	4.63%	29.00%	0.1000	0.1444	4.44%	44.44%	0.1176	0.1617	4.41%	37.47%
POI	0.7377	0.7605	2.27%	3.08%	0.8461	0.8260	-2.02%	-2.39%	0.7743	0.7769	0.27%	0.34%
Synapse	0.5077	0.5549	4.72%	9.29%	0.2275	0.2720	4.45%	19.55%	0.2928	0.3466	5.39%	18.39%
Velocity	0.4533	0.4986	4.53%	9.98%	0.2099	0.2204	1.05%	5.01%	0.2611	0.2835	2.25%	8.60%
Xalan-J	0.6087	0.6643	5.57%	9.14%	0.4242	0.4814	5.72%	13.47%	0.4779	0.5414	6.35%	13.28%
Xerces	0.4295	0.4183	-1.12%	-2.60%	0.1275	0.1362	0.87%	6.80%	0.1743	0.1808	0.65%	3.72%
Average	0.4556	0.5029	**4.73%**	**12.90%**	0.2922	0.3175	**2.52%**	**19.10%**	0.3144	0.3498	**3.54%**	**18.20%**

TABLE VII exhibits the performance increases after CE-AST feature being added, where the results are the mean values of three classifiers with six different feature suites. As the table shows, the performance indexes gain a different degree of growth on 12 Java datasets. Roughly, the projects of Log4j, Xalan-J and Ant obtain more absolute increases, meanwhile Lucence, POI, and Xerces are opposite. There is a little different in case of relative comparison, e.g., the growths on Ivy and PBeans are more remarkable. On average, CE-AST contributes an average of 4.73% absolute increase in Precision, 2.52% in Recall and 3.54% in F1, where the relative increases are 12.90%, 19.10% and 18.20% correspondingly. Overall, as seen from TABLE VII, CE-AST does bring improvement in most defect prediction tasks and plays a positive role on the 12 investigated projects, no matter in terms of relative comparison or absolute comparison.

CE-AST is complementary to traditional feature sets in defect prediction, and can improve the prediction performance on different defect datasets.

V. CONCLUSION

The cross-entropy of a sequence measures its likelihood of appearance in the corpus. Common sequences usually own higher probability of occurrence, that is, lower cross-entropy. In recent years, a series of work have used cross-entropy for code measurement and found that defective codes tend to be more entropic. Based on this finding, this paper introduces cross-entropy as a new feature for defect detection. Considering that the code's AST contains rich semantic and structural information, we designed a code feature CE-AST, i.e., the cross-entropy value of the sequence of AST nodes, for better defect prediction. To evaluate the effectiveness of CE-AST, we conduct several experiments on a widely used dataset of 12 Java open-source projects. As the results show, the CE-AST feature owns more discrimination power for defect-proneness classification than 45% of twenty common traditional features, and does bring improvement of 4.7% in Precision, 2.5% in Recall and 3.5% in F1 by average to prediction models. In conclusion, these findings suggest that CE-AST is useful to predict software defects and complementary to existing code feature sets.

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] 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.

[2] 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.

[3] S. Y. Lee, D. Li, and Y. Li, "An investigation of essential topics on software fault-proneness prediction," in *ISSSR'16*: *the International Symposium on System and Software Reliability*, pp. 37-46, 2016.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S Khudanpur, "Recurrent neural network based language model," in *INTERSPEECH'10*: *Annual Conference of the International Speech Communication Association*, pp. 1045-1048, 2010.

[21] T. Mikolov, W. T. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 746-751, 2013.

[22] 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.

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

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

[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] 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.

[27] 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.

[28] 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.

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

[30] 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.