A Framework based on Deep Neural Network for Ranking-oriented Software Defect Prediction

Jiapeng Dai[1], Xiaoxing Yang[1,2,*], Bingding Huang[1,*], and Xiaofen Lu[2]
[1]College of Big Data and Internet, Shenzhen Technology University, Shenzhen, 518118, China
[2]Guangdong Provincial Key Laboratory of Brain-Inspired Intelligent Computation,
Department of Computer Science and Engineering, Southern University of Science and Technology,
Shenzhen, 518055, China
2110416003@stumail.sztu.edu.cn, yangxiaoxing@sztu.edu.cn, huangbingding@sztu.edu.cn, luxf@sustech.edu.cn
*corresponding author

Abstract—Software systems are getting larger and more complex than ever before. In order to improve software reliability, software defect prediction is applied to assist developers in bug discovery. The ranking-oriented software defect prediction aims to rank software modules according to the predicted defect counts. However, existing ranking-oriented defect prediction models are constructed based on traditional hand-crafted features, which might overlook the rich syntactic information buried inside the source codes. In this paper, we propose a universal deep learning-based framework called United Deep Network for ranking-oriented software defect prediction. This framework utilizes deep neural networks to automatically generate features from source code with the syntactic and structural information preserved, and it can combine extracted features with traditional hand-crafted features in order to take advantage of both kinds of features to construct prediction models. Experimental results over 29 sets of data show the good performance of the proposed framework for building ranking-oriented defect prediction models.

Keywords–ranking-oriented software defect prediction; deep neural network; convolutional neural network; recurrent neural network

1. Introduction

The ever-increasing complexity of modern software systems has enhanced the demands for software reliability. Software defects are flaws or deficiencies in software that prevent it from working as expected and bring about software failures [41]. Software Defect Prediction (SDP) is a technique to predict code areas that potentially contain defects [15]. The code areas could be files, methods, classes or packages. Most existing SDP studies [11], [15], [23], [30], [41] consider SDP as a binary classification problem, which focuses on predicting whether defects exist in a specific area of source code or not [23]. Meanwhile, some SDP studies [27], [38], [39] take the number of defects within software modules into account and rank software modules according to the predicted number of defects. The ranking result reflects the priority of code inspection or unit testing, thus serving as a useful tool for determining the order in which code should be inspected. This paper attempts to address the problem of ranking-oriented defect prediction, and focuses on file-level prediction. Typical SDP is mainly composed of three steps [27]: (1) collecting or extracting features from source files; (2) training a prediction model based on the obtained features and bug information using model construction methods such as traditional machine learning methods; (3) and applying the trained model to predict probability or quantity of software defects in new code areas. In general, previous studies attempt to tackle SDP in two research directions: to design or generate more expressive features; or to construct better models. In order to obtain representative features, many previous studies [15], [21] utilized various discriminative hand-crafted features to capture holistic software characteristics, such as Halstead features based on the number of operators and operands [6], McCabe features based on dependencies [16], Chidamber and Kemerer (CK) features for the object-oriented programs [8], and so on. However, since programming languages are designed with well-defined syntax, traditional hand-crafted features may overlook the rich syntactic and local structural information that are buried deeply inside program's abstract syntax trees (ASTs) [33].

In recent years, deep learning has made remarkable strides and achieved significant progress in virtually every sphere of research, such as computer vision and natural language processing. Motivated by these huge success, some studies [15], [23] attempted to extract features and predict defects directly from software ASTs using convolutional neural networks (CNN). They have made progress using the extracted features over PROMISE Source Code (PSC) dataset [15] and Simplified PROMISE Source Code (SPSC) dataset [23], but these studies focused on a binary classification task rather than a ranking-oriented SDP problem.

In order to construct better models, researchers compare multiple methods to select or directly design proper methods. For instance, Yu et al. [39] compared various methods including pointwise, pairwise and listwise approaches for ranking-oriented SDP over public datasets including AEEEM [5] and Eclipse [43], and the authors found that when using module as effort, random forest regression performed best under cross-release setting; and Yang et al. [36] compared ridge and lasso regression methods with existing methods and found that ridge regression [36] and Random Forest Regression (RFR) [35] could achieve better results under cross-release setting. With

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00016

the popularity of deep learning, some researchers (such as Lei et al.' work [27] and Meetesh and Pradeep's work [22]) also applied deep learning methods for solving ranking-oriented SDP problems. However, these studies extracted features from traditional hand-crafted features instead of source code.

Due to the lack of deep learning to extract new features directly from source code for ranking-oriented SDP problems, we attempt to fill this gap in this study. To be specific, we propose a ranking-oriented SDP framework called *United Deep Network* (UDN), which makes use of both deep syntactic features and traditional hand-crafted features. Specifically, inspired by [15], we extract syntactic features from software ASTs using deep neural network like CNN and RNN, and concatenate the extracted deep syntactic features with the traditional hand-crafted features in order to take full advantage of both rich local structural features and precise holistic features of source codes. We then feed the combined features to model construction methods such as Random Forest Regression (RFR) to generate ranking-oriented SDP models.

The main contributions of this paper are as follows.

- This paper first extracts features from source code using deep learning for ranking-oriented SDP.
- This paper proposes a simple and universal ranking-oriented SDP framework which combines features extracted from ASTs using deep neural network like CNN and RNN with traditional handcrafted features, in order to fully utilize both rich local syntactic features and precise holistic features of source codes to construct ranking-oriented SDP models. Furthermore, this paper compares different model construction methods based on the combined features, in order to give a comprehensive comparison.
- This paper also compares different deep neural network (DNN) frameworks, including GRU [2] and ResNet [7].

2. Background

As mentioned by some previous work [15], [23], a typical file-level software defect process is composed of multiple steps, as shown in Figure 1.

The first step is to preprocess data and labels according to the demands and limitations of the problem to solve and the methods to use. Labels should be non-negative integers to indicate number of defects in a file for the ranking-oriented SDP task (and integers should be binarized to indicate whether bugs exist in a file for the binary defect classification task). As in some CNN-based SDP research [15], [23], source codes of both training files and testing files would be parsed into Abstract Syntax Trees (ASTs), then select representative nodes on ASTs to form token vectors. Thus each source code file would be transferred to a token vector before being fed into the following encoding phase.

The second step is to extract features from source files using traditional hand-crafted methods or deep learning models. In general, there are two categories of traditional hand-crafted features: code metric features (e.g., McCabe features [16], Halstead feature [6] and CK features [8]), and process metric features (e.g., change histories). Deep-learning-based feature extraction utilizes deep learning models like deep belief network (DBN) [31] and CNN [15], [23], [27] to extract features. Some existing classification-oriented deep-learning-based methods extract file features from ASTs, while other studies [22], [27] extract features from selected hand-crafted features.

In the third step, the obtained features are utilized as training instances to build predictive model using various machine learning algorithms, such as Logistic Regression, Linear Regression, Support Vector Machine (SVM), Naive Bayes and Decision Trees.

Finally, new instances would be fed into the trained model to generate predicted defect information. Prediction results would be evaluated using some metrics like F-measure (also F1 score) for classification-oriented SDP or fault-percentile-average (FPA) metric [32] for ranking-oriented SDP, in order to measure model performance.

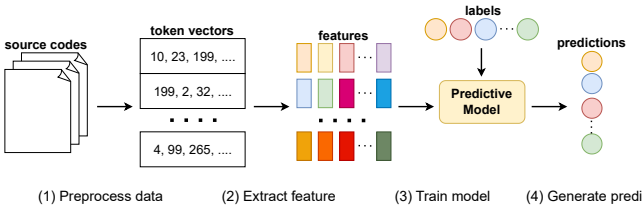

Figure 1. A typical software defect prediction process.

In practice, the SDP problem could be categorized into within-project-defect-prediction (WPDP) and cross-project-defect-prediction (CPDP), based on whether both training and test instances come from the same software project. Similarly, WPDP could be further categorized into within-version-WPDP and cross-version-WPDP, according to whether all the instances come from the same version of the project. In this work, we focus on the ranking-oriented cross-version-WPDP problem. Following the conventions of previous research [15], within a project, we use instances from an older version for training, and instances from a newer version for testing.

3. Approach

In this section, we elaborate our proposed United Deep Network (UDN), a DNN-based framework for ranking-oriented SDP which can generate syntactic features from source codes, and can combine the generated features with traditional hand-crafted features. The overall workflow of UDN is illustrated with Figure 2.

3.1 Preprocessing Source Code

Source code is essentially a sequence of text and requires further conversion to the representations that machine can directly process. Following conventions of natural language processing (NLP), text sequence data like source code requires some preprocessing before training, and the typical preprocessing procedure includes two steps: *tokenization* and *token encoding*. Tokenization is the process of segmenting a sequence of text into individual units, and the units here are called the

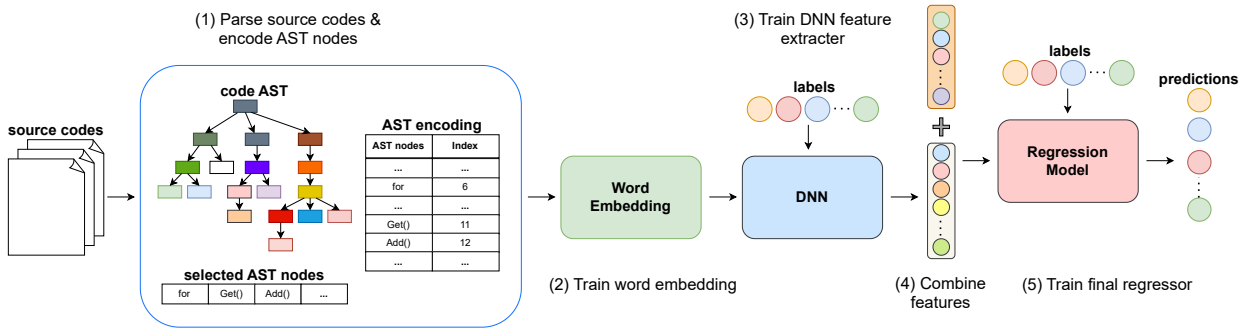

Figure 2. The overall workflow of our proposed UDN.

tokens of the sequence. For file-level tokenization, each source code file would be transferred to a vector of tokens. We can perform tokenization on different levels of granularities. For software programs, according to [23], possible granularities of tokenization could be character-level, token-level, AST-node-level, tree-level, etc. As analyzed in [26], AST-node-level tokenization is the optimal option for building program representation, which preserves both syntactic and structural information of programs. There are numerous open-source libraries available for parsing source code to ASTs, which provides convenient workflow to perform AST-node-level tokenization. For example, some previous SDP research [15], [23] take advantage of an open-source Python package called javalang[1] to get their Java source code tokenized. The AST nodes would be selected automatically according to certain filtering rules and output as sequential tokens, in order to serve as the inputs for the subsequent models.

In practice, a vast majority of machine learning frameworks require numerical vectors as inputs, so the generated token vectors cannot be directly sent to these models. To solve this problem, following [15], [23], we build a mapping to encode each token to an associated integer. Each token corresponds to a unique integer identifier which ranges from 1 to number of token types. Thus tokens can be distinguished from each other via their unique identifiers. Additionally, models like CNN require input vectors to share the equal length while text sequences like source code often differ in their lengths. In the event of this issue, we simply pad zeros at the end of each integer vector, making all of them matching the length of the longest vector. The padded zeros would not disrupt the encoding because the identifiers with meaning start from one. Following common practices in NLP domain, we delete infrequent tokens which appear less than three times by encoding them to zero.

3.2 Word Embedding

In NLP, word embedding is a technique that transfers a word to a corresponding real-valued vector that encodes the meaning of the word. The embedded vectors capture the similarity among words in such a way that words that are closer in the vector space are expected to be similar in meaning [12]. Prior to the proposal of word embedding, words are used to be represented in either the naive integer form as discussed in the former subsection or in the one-hot form with the size of corpus, and both representations fail to express the similarity among words precisely.

Word2vec [18], [19] is one of the most commonly used word embedding techniques and it uses neural network to learn word associations from a large corpus of words. Word2vec is one of those unsupervised or self-supervised learning techniques which attempt to train models without utilizing any hint of labels. In general, the word2vec family contains two types of architectures, namely *Continuous Bag of Words* (CBOW) [18] and *skip-gram* [19]. In both architectures, word2vec takes both current words and a sliding window of context words surrounding the current words into account as it iterates over the entire corpus. In CBOW architecture, the model predicts current word from the surrounding context words. In skip-gram architecture, the model utilizes current word to predict context words within the sliding window. In general, CBOW is faster but skip-gram does a better job for infrequent words [19]. We use skip-gram as our word embedding architecture. In the skip-gram model, each word plays two roles: as a *center* word and as a *context* word. Thus each word has two d-dimensional-vector representations for calculating conditional probabilities. For any word w_i with index i, we denote its two vector representations by $v_i \in \mathbb{R}^d$ and $u_i \in \mathbb{R}^d$ for center word and context word, respectively. According to [42], the conditional probability of generating any context word w_o given the center word w_c can be modeled by the softmax on vector dot product:

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \tag{1}$$

where $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ is the corpus index set. Word2vec model is a shallow, two-layer neural network, whose parameters are the center word vector and context word vector for each word in the corpus. Given a text sequence of length T, where the word at time step t is denoted as $w^{(t)}$. Assume that

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

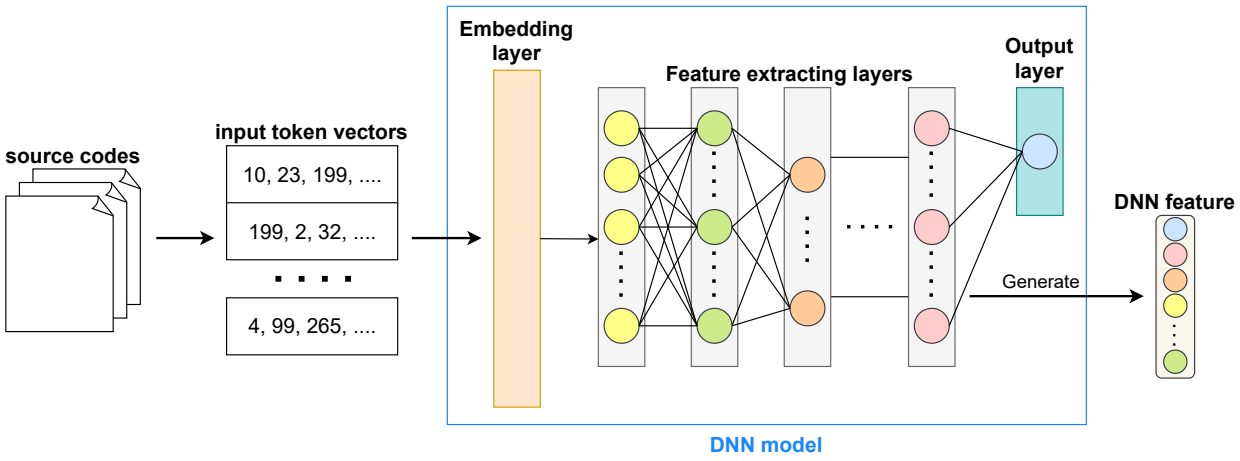

Figure 3. Basic DNN architecture of our proposed UDN.

context words are generated independently given any center word. For window size l, we train the model by minimizing the following logarithmic loss function:

$$-\sum_{t=1}^{T}\sum_{-l\le j\le l,\ j\neq 0}\log P(w^{(t+j)}\mid w^{(t)}), \tag{2}$$

Following the conventional training process, we use the negative sampling technique [19] to optimize the computational complexity of training word2vec. We train word embedding using all training instances prior to training DNN models and store the trained embedding for future use. Before evaluating on a new dataset, we simply replace the previously unseen tokens with a specific unknown token.

3.3 Building Deep Neural Network

The basic DNN architecture of the proposed UDN and training procedure is illustrated in Figure 3. We train our DNN model using training instances that are preprocessed into integer ASTs tokens. Our DNN model starts with an embedding layer, which turns positive integer indexes into real-valued vectors of fixed size, and ends with a dense (linear) layer as the output layer of the DNN model. We replace the embedding layer with the pretrained word2vec embedding from the former subsection before training and then freeze the weights of the embedding layer. We set the output size of the final dense layer to 1 in order to generate defect count prediction and train the DNN model using gradient descent based optimizing technique. It should be noted that the dense output layer here is used to optimize network parameters in order to improve the quality of the extracted features, so the outputs before the final dense layer in the DNN model are considered to be the extracted features.

For the purpose of validating the universality of aforementioned feature combining strategy, we implement three different DNN models as backbone feature extracting models, which are based on the widely adopted VGG [29], ResNet [7] and GRU [2]. Our implementation is based on *Pytorch* [24] and the detailed design will be discussed in the remainder of this subsection.

3.3.1 Visual Geometry Group (VGG)

VGG is a widely recognized convolutional neural network architecture that achieved remarkable performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014 [28]. It is originally designed for 2D image recognition. Its key feature is the use of 3×3 Conv2d (2D convolutional) filters throughout the entire network, allowing for deeper architectures with fewer parameters. The architecture's simplicity and effectiveness make it a popular benchmark and a base for further network development. For SDP problem, each input token of the source codes would be transferred to a 1D vector. To adapt such change in data dimension, all the layers tailored for 2D data in VGG as well as other CNN architectures, such as convolutional layers and pooling layers, should be replaced by their 1D variations.

As illustrated in Figure 4, the VGG structure used in our UDN is composed of multiple VGG blocks, followed by an MLP (Multi-layer Perceptron). Each VGG block is composed of a Conv1d (1D convolutional) layer, a BatchNorm (batch normalization) layer, a ReLU activation layer and a MaxPool (max pooling) layer in sequence. The batch normalization layer is used to mitigate the internal covariate shift inside deep networks by performing data normalization [10], which is neglected by almost all previous CNN-based SDP research. The MLP at the end of the VGG is composed of several dense layers, and similar to VGG block, a BatchNorm layer, a ReLU activation layer and a Dropout layer (drop rate=0.5) are placed between two dense layers. For this study, we used a VGG model composed of three VGG blocks and a four-layer MLP. For the three VGG blocks, the three Conv1d layers share almost the same settings (kernel size=3, stride=1, padding=1) except for output channel size (64, 32, 32, respectively), and all the three MaxPool layers are set to be the same (kernel size=2, stride=2). For the MLP layers, the hidden size of the dense layers are set to 64 except for the last two of them (20

for feature extraction and 1 for model output, respectively).

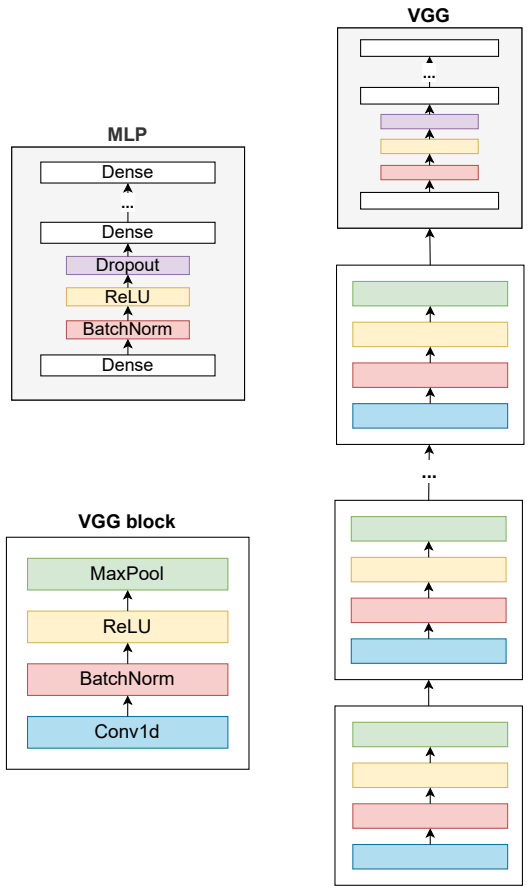

Figure 4. The VGG model structure for UDN.

3.3.2 Residual Network (ResNet)

ResNet is a deep learning neural network architecture that was introduced in 2015 by He et al [7]. It is known for its unique structural characteristic of using residual blocks, which are made up of multiple layers, to overcome the vanishing gradient problem that can occur in very deep neural networks. In a residual block, the output of one layer is added to the output of a previous layer, allowing the network to better propagate gradients through the network and improve its accuracy. Its ability to train very deep networks has made it a popular choice for many applications in computer vision and beyond. Hence, we attempt to apply the ResNet architecture to SDP problem. Similar to VGG, ResNet was originally tailored for 2D image data. Therefore, we replace the layers associated with 2D inputs like convolutional layer and pooling layer to their 1D versions.

The ResNet structure that we use is depicted in Figure 5. In general, ResNet could be devided into three parts from front to back. (1) The first part consists of a sequence of Conv1d layer (kernel size=7, stride=2, padding=3, output channel=64), BatchNorm layer, ReLU activation layer and MaxPool layer (kernel size=3, stride=2). (2) The second part is composed of several residual blocks. Within each residual block, a series of Conv1d layer, BatchNorm layer, ReLU actvation layer, Conv1d layer and BatchNorm layer are used. At the end of the block, the BatchNorm output is added with the block input with a skip connection before being fed into the final ReLU activation layer. Two Conv1d layers have similar settings (kernel size=3, padding=1) except for the strides (2 for the first Conv1d, 1 for the second), and their hidden channel size would be the same as the block output channel size. For this study, we use a simplified ResNet with three residual blocks (output channels=64, 32, 32). A pointwise Conv1d layer with kernel size 1 is used on the skip connection in the latter two blocks in order to align channel of block inputs with the BatchNorm layer outputs before adding. Further implementation details can be found in [42]. (3) The last part of ResNet is a group of an MLP and an AdaptiveAvgPool (global average pooling) layer gathering global features before MLP. In the end, we use a four-layer MLP with the same settings as in VGG model described before.

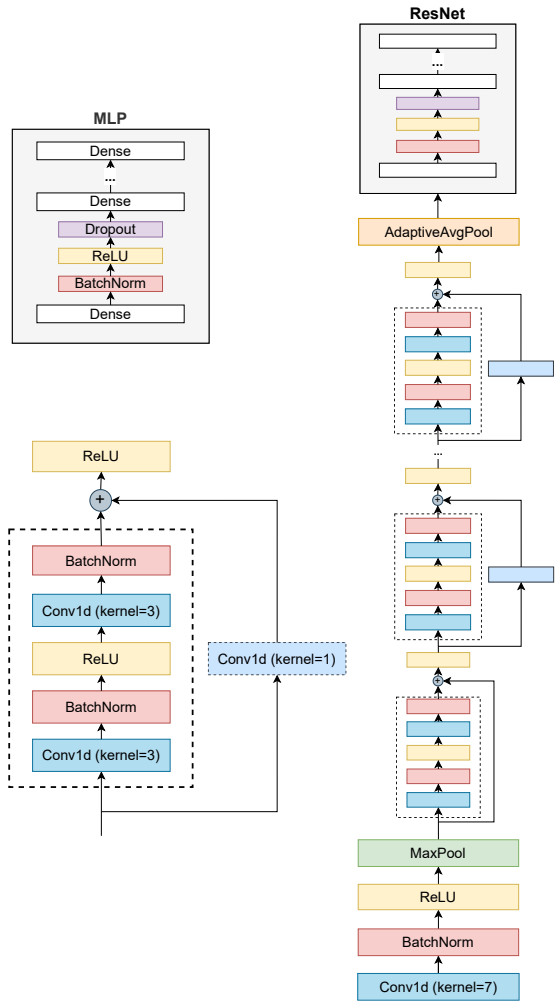

Figure 5. The ResNet model structure for UDN.

3.3.3 Gated Recurrent Unit (GRU)

GRU is a type of recurrent neural network (RNN) introduced by Cho et al. [2] in 2014 as a simplified version of the long short-term memory (LSTM) network [9]. The GRU cell contains two gates, a reset gate and an update gate, that control the flow of information in the network. The reset gate decides how much of the previous hidden state to forget, while the update gate decides how much of the new input to incorporate into the current hidden state. The use of the reset and update gates allows the GRU to effectively capture long-term dependencies in sequential data while avoiding the vanishing gradient problem that can occur in traditional RNNs. This makes GRU a powerful tool for a wide range of sequence modeling tasks.

We use the GRU model shown in Figure 6, which is composed of several GRU layers and an MLP. We concatenate the outputs from the final GRU layer across all time steps and utilize them as the inputs of MLP. For this study, we implement a model including two GRU layers with hidden size of 64 and a three-layer MLP with dense hidden size of 64, 20 and 1, respectively.

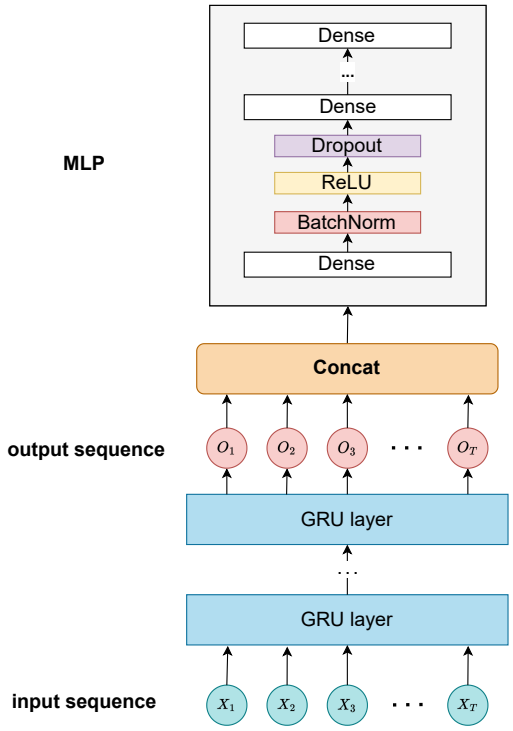

Figure 6. The GRU model structure for UDN.

3.4 Combining Traditional Features

In conventional defect prediction methods, traditional hand-crafted features such as complexity metrics and process metrics are shown to be informative in distinguishing buggy code [15]. They are basically some numbers which describe certain characteristics of codes from a holistic perspective. Typical hand-crafted code features include Lines of Code (LOC), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), and McCabe complexity measures (Max CC and Avg CC), etc. To take advantage of these information, we directly concatenate the DNN-learned feature vectors with traditional hand-crafted feature vectors. Finally, the combined feature vectors are fed into the subsequent regression model in order to generate defect count prediction.

3.5 Training Prediction Models

Yang et al. [36] have demonstrated the effectiveness of ridge regression (RR) for ranking-oriented SDP tasks. According to the comparison results of 34 algorithms [39], when using module as effort, random forest regression (RFR) performed best under cross-release setting. Therefore, we use RR and RFR as our final prediction generators. For comparison, we use classical Linear Regression (LR) as the baseline. All regression methods are implemented using the well-known open-source machine learning library scikit-learn [25]. Parameters of regression methods are set according to previous work [35], [36].

3.6 Other Implementation Details

- Mean Squared Error (MSE) is used as loss function when training DNN models. Our DNN models are trained in the batch size of 32 and learning rate of 0.001 with the Yogi optimizer [40], an improved version of Adam optimizer [13], which alleviates the issue of Adam potentially diverging due to poor variance control. We use the existing Yogi implementation from an open-source library[2].
- In order to prevent over-training, we adopt L2 Regularization with penalty factor of 0.001 and early stopping strategy when training DNN models. We set the training epochs of VGG, ResNet and GRU to 18, 15, 18, respectively.

4. Evaluation

4.1 Evaluation Metrics

Following the previous ranking-oriented SDP research [27], [38], [39], we use fault-percentile-average (FPA) metric [32] to evaluate the prediction results of our proposed UDN framework. FPA is essentially the average of propotions of actual defects in the top m modules to the whole defects [35]. Specifically, considering k modules listed in increasing order of predicted defect number as $y_1, y_2, \ldots, y_k$, and assuming that n_i is the actual defect number in the module i, $n = n_1+n_2+\cdots+n_k$ is the total number of defects. The proportion of the actual defects in the top m predicted modules to the whole defects is

$$\frac{1}{n}\sum_{i=k-m+1}^{k} n_i \tag{3}$$

Then the FPA [32] of the k ranked modules is defined as:

[2]https://github.com/jettify/pytorch-optimizer

$$\frac{1}{k}\sum_{m=1}^{k}\frac{1}{n}\sum_{i=k-m+1}^{k}n_i \tag{4}$$

A higher FPA means a better ranking and the module with the most defects comes first. It is worth noting that although the values of FPA lie between 0 and 1, the optimal FPA value (when predicted results perfectly match the labels) are consistently smaller than 1.

It would be not intuitive enough to compare different modules via the originally defined FPA. To tackle the issue, we have further defined a relative-FPA (rFPA) metric, which is the ratio of the actual model's FPA to the FPA of the ideal model (whose predicted results are the actual labels), making the rFPA of the best predictions equal to 1.

4.2 Dataset

For this study, we use the modified version of PROMISE source code (PSC) dataset[3] provided by Pan et al. [23], which contains 14,066 files from 41 versions of 12 projects and is slighlty different from the original PSC dataset [15] whose given download link is currently invalid. In this work, we focus on the ranking-oriented cross-version-WPDP task as stated before. We use an older version as training set and a newer version as testing set within a project. The 29 training groups used in our experiments are listed in Table I. It is noticeable that, in cross-version WPDP task, the data scale is relatively small and a training set might be smaller than its test set, which poses significant challenges for this task.

Additionally, the 20 hand-crafted features of each source file that we use are provided by the dataset. Further details of the hand-crafted features and the PROMISE dataset could be found in the original paper [15].

4.3 Research Questions

In this paper, we attempt to investigate the following questions.

- **What can our UDN bring? And Which DNN model can extract the most effective features directly from source code?**
  In order to answer this question, we use hand-crafted features as the baseline, and we compare three combinations of hand-crafted features and extracted features by UDN respectively using VGG, ResNet and GRU as backbone DNN models. We use three regression methods to construct models.
- **Which kind of features is most effective?**
  We choose the best model construction algorithm of LR, RR, and RFR according to the above experiment, and the best backbone DNN model, in order give a comprehensive comparison of multiple kinds of features: hand-crafted features, extracted features, and combination of both features.

[3]https://github.com/penguincwarrior/CNN-WPDP-APPSCI2019

TABLE I
TRAIN-TEST GROUPS OF PSC DATASET [23] USED IN OUR EXPERIMENTS.

Project	Train Ver.	Test Ver.	#Train Files	#Test Files
Ant	1.3	1.4	124	177
	1.4	1.5	177	278
	1.5	1.6	278	350
	1.6	1.7	350	741
Camel	1.0	1.2	339	595
	1.2	1.4	595	847
	1.4	1.6	847	934
Ivy	1.1	1.4	111	241
	1.4	2.0	241	352
JEdit	3.2	4.0	260	281
	4.0	4.1	281	266
	4.1	4.2	266	355
	4.2	4.3	355	487
Log4j	1.0	1.1	119	104
	1.1	1.2	104	194
Lucene	2.0	2.2	186	234
	2.2	2.4	234	330
Pbeans	1.0	2.0	26	51
Poi	1.5	2.0	235	309
	2.0	2.5	309	380
	2.5	3.0	380	438
Synapse	1.0	1.1	157	205
	1.1	1.2	205	256
Velocity	1.4	1.5	195	214
	1.5	1.6	214	229
Xalan	2.4	2.5	676	754
	2.5	2.6	754	875
Xerces	init	1.2	162	436
	1.2	1.3	436	446

4.4 Results and Discussions

4.4.1 *Performance of Various Model Combinations*

In this experiment, we treat models using solely hand-crafted features as the baseline, and compare their ranking performance with our models using the combined features, in order to investigate whether the proposed approach can lead to performance improvement. By comparing three backbone DNN models (VGG, ResNet and GRU) and three model construction

methods (LR, RR and RFR), we want to find the best backbone DNN model and the best model construction methods. LR is a classical regression method, and RR and RFR have been demonstrated as the best methods for constructing ranking-oriented SDP models according to some large-scale study [36], [39]. Therefore, we attempt to implement our method based on these previously validated regression models.

The testing FPA and rFPA (in the brackets) results of our proposed UDN framework with various model combinations on the modified PSC dataset are displayed in Table II. The columns of "regressor-only" record results of the model constructed by the corresponding regressor using only hand-crafted features. For example, the column of "LR-only" records results of LR using only hand-crafted features. The columns of "DNN+regressor" record results of the model constructed by the corresponding regressor using combined features of hand-crafted features and extracted features by the corresponding DNN method. For example, the column of "VGG+LR" records results of LR using combined features of hand-crafted features and extracted features by VGG.

According to the results shown in Table II, the utilization of deep neural network, in combination with various regression models, can bring an enhancement of ranking performance. All combinations have improved performance to varying degrees, compared with the baseline performance. Among these combination, the best average ranking performance of 0.715 FPA (79.9% rFPA) is achieved using the combination of ResNet and RFR, improving the baseline performance of solely RFR by 0.014 FPA (1.1% rFPA), and by 0.046 FPA (4.9% rFPA) compared with the baseline performance of solely simple LR.

From the obtained results of each model combination, we can conclude that the choice of the final regression model determines the maximum achievable performance and affects the effectiveness of the proposed model combination strategy. We investigate three commonly-used regression models in ranking-oriented SDP task and find that their baseline performance follows the order of LR (0.669 FPA) $<$ RR (0.694 FPA) $<$ RFR (0.701 FPA). The performance of their current best combinations with different deep models also reflects such trend, where VGG+LR (0.672 FPA) $<$ GRU+RR (0.701 FPA) $<$ ResNet+RFR (0.715 FPA). These results indicate that different regression models may require different deep models to achieve optimal performance.

Meanwhile, it can be found that such model combination strategy leads to slightly different levels of improvement for different regression models. From the current results, we could see that ResNet+RFR improves the baseline RFR by 0.014 FPA, while ResNet+LR only improves the baseline LR by 0.001 FPA and ResNet+RR only improves the baseline RR by 0.003 FPA. It might be the case that LR and RR are both linear models and have limitations in dealing with the significant distribution differences between deep features and hand-crafted features due to the potential multi-collinearity issue, while nonlinear models like RFR do not suffer from such drawback.

4.4.2 *Comparing Different Kinds of Features*

According to Table II, models using RFR as the final regression model achieves the best performance comparing to the other two regressors. In order to investigate the effectiveness of the proposed feature combination strategy, we compare the performance of combined features comparing to other feature strategies on all three DNN models with RFR as the final regressor. The feature strategies to be compared includes (1) using hand-crafted features only, (2) using DNN-extracted features only, and (3) using combined features. The comparing results are listed in Table III.

According to Table III, the feature combing strategy achieves the best overall performance with all kinds of DNN backbone, comparing to the hand-crafted-only or DNN-extracted-only feature strategy. Comparing to the baseline overall performance of hand-crafted-only features (0.701 FPA), all the DNN-extracted-only features deliver poorer overall performances (0.698, 0.609, 0.697 for VGG, GRU and ResNet respectively). Such results indicate features extracted by deep neural models do not necessarily guarantee a performance boost comparing to the hand-crafted statistical features. Considering the rather small data scale of the commonly used SDP dataset like PROMISE [15], deep models are susceptible to the potential issue of overfitting. Meanwhile, the combined features with all three DNN backbones outperform the baseline hand-crafted-only feature in the overall ranking performance (0.709, 0.712, 0.715 for VGG, GRU and ResNet respectively). It should be noted that the combined features with ResNet backbone and RFR regressor achieves the best overall FPA (0.715), while GRU-backbone combined feature wins most of the train groups (10 out of 29) and achieves relatively similar overall performance (0.712 FPA) with the best record. These results provide strong evidence of the universality of the proposed feature combining strategy, that all kinds of deep neural models like CNN and RNN can benefit from it.

5. Related Work

5.1 Software Defect Prediction

Software defect prediction (SDP) is an active research field in the area of software engineering [11], [17], [20], [30]. Most SDP methods tend to consider SDP as binary classification problem, which focuses on predicting whether defects exist in a specific area of source code or not [11], [15], [23], [30], [41]. In recent years, there have been some ranking-oriented studies [27], [38], [39] that took number of defects within software modules into account and ranked software modules by the predicted number of defects. Many early studies [11], [15], [21] focused on manually designing new discriminative features or new combinations of features from labeled historical defect data in order to capture holistic software characteristics, for example, Halstead features [6], McCabe features [16], and Chidamber and Kemerer (CK) features [8], etc.

TABLE II
FPA AND RFPA (IN THE BRACKETS) RESULTS OF OUR PROPOSED UDN WITH VARIOUS MODEL COMBINATIONS ON THE MODIFIED PSC DATASET, WHERE "XX-ONLY" MODELS ONLY USE HAND-CRAFTED FEATURES.

Train	Group	LR-only	VGG+LR	GRU+LR	ResNet+LR	RR-only	VGG+RR	GRU+RR	ResNet+RR	RFR-only	VGG+RFR	GRU++RFR	ResNet+RFR
Ant	1.3-1.4	0.544 (60.1%)	0.535 (59.1%)	0.544 (60.2%)	0.513 (56.6%)	**0.577 (63.8%)**	0.568 (62.8%)	0.573 (63.4%)	0.539 (59.6%)	0.577 (63.8%)	0.543 (60.0%)	0.549 (60.7%)	0.564 (62.3%)
	1.4-1.5	0.638 (66.6%)	0.571 (59.7%)	0.602 (62.9%)	0.636 (66.4%)	0.665 (69.5%)	0.665 (69.5%)	0.677 (70.8%)	**0.682 (71.3%)**	0.606 (63.3%)	0.512 (53.5%)	0.574 (60.0%)	0.584 (61.0%)
	1.5-1.6	0.777 (84.8%)	0.658 (71.7%)	0.724 (79.0%)	0.692 (75.5%)	**0.791 (86.3%)**	0.755 (82.3%)	0.787 (85.8%)	0.746 (81.4%)	0.725 (79.1%)	0.737 (80.4%)	0.738 (80.4%)	0.749 (81.7%)
	1.6-1.7	0.794 (85.4%)	0.767 (82.5%)	0.765 (82.3%)	0.778 (83.6%)	0.804 (86.4%)	0.794 (85.3%)	**0.805 (86.6%)**	0.797 (85.6%)	0.805 (86.5%)	0.798 (85.8%)	0.794 (85.4%)	0.800 (86.0%)
Camel	1.0-1.2	0.615 (69.2%)	0.525 (59.0%)	0.512 (57.6%)	0.577 (64.9%)	0.645 (72.5%)	0.610 (68.6%)	0.642 (72.2%)	0.634 (71.3%)	**0.658 (73.9%)**	0.639 (71.9%)	0.629 (70.7%)	0.642 (72.2%)
	1.2-1.4	0.678 (71.4%)	0.801 (84.2%)	0.800 (84.2%)	0.784 (82.5%)	0.756 (79.6%)	0.799 (84.1%)	0.822 (86.5%)	0.783 (82.4%)	0.778 (81.9%)	0.814 (85.6%)	**0.827 (87.0%)**	0.805 (84.7%)
	1.4-1.6	0.713 (75.6%)	0.742 (78.6%)	0.770 (81.6%)	0.787 (83.4%)	0.737 (78.1%)	**0.800 (84.8%)**	0.782 (82.8%)	0.791 (83.8%)	0.743 (78.8%)	0.778 (82.5%)	0.794 (84.1%)	0.778 (82.5%)
Ivy	1.1-1.4	0.728 (74.9%)	0.833 (85.8%)	0.834 (85.9%)	0.826 (85.0%)	0.744 (76.6%)	0.802 (82.6%)	0.812 (83.7%)	0.835 (86.0%)	0.746 (76.8%)	0.787 (81.0%)	0.808 (83.3%)	**0.836 (86.1%)**
	1.4-2.0	0.470 (49.2%)	0.600 (62.7%)	0.668 (69.8%)	0.569 (59.6%)	0.577 (60.3%)	0.683 (71.5%)	0.703 (73.5%)	0.707 (74.0%)	0.723 (75.7%)	0.768 (80.3%)	0.770 (80.6%)	**0.772 (80.7%)**
JEdit	3.2-4.0	0.847 (89.8%)	0.853 (90.4%)	0.862 (91.5%)	0.787 (83.5%)	0.845 (89.6%)	0.866 (91.8%)	**0.876 (92.9%)**	0.851 (90.3%)	0.854 (90.6%)	0.862 (91.4%)	0.867 (92.0%)	0.864 (91.6%)
	4.0-4.1	0.801 (85.9%)	0.833 (89.3%)	0.815 (87.4%)	0.796 (85.4%)	0.805 (86.3%)	0.848 (90.9%)	0.796 (85.4%)	0.805 (86.4%)	0.839 (89.9%)	0.843 (90.4%)	0.841 (90.2%)	**0.855 (91.7%)**
	4.1-4.2	0.872 (90.7%)	0.826 (85.9%)	0.593 (61.7%)	0.836 (87.0%)	0.874 (90.9%)	0.871 (90.5%)	**0.879 (91.5%)**	0.876 (91.1%)	0.859 (89.3%)	0.864 (89.8%)	0.869 (90.4%)	0.873 (90.8%)
	4.2-4.3	0.592 (59.8%)	0.680 (68.7%)	0.723 (72.9%)	0.656 (66.2%)	0.588 (59.3%)	0.584 (58.9%)	0.668 (67.4%)	0.561 (56.6%)	0.616 (62.2%)	**0.773 (78.1%)**	0.704 (71.1%)	0.692 (69.8%)
Log4j	1.0-1.1	0.757 (84.8%)	0.771 (86.3%)	0.744 (83.4%)	0.749 (83.9%)	0.787 (88.2%)	**0.812 (91.0%)**	0.767 (86.0%)	0.801 (89.8%)	0.748 (83.8%)	0.780 (87.4%)	0.787 (88.1%)	0.775 (86.8%)
	1.1-1.2	0.558 (87.4%)	0.541 (84.8%)	0.554 (86.8%)	0.544 (85.3%)	0.558 (87.3%)	0.558 (87.3%)	0.561 (87.8%)	0.554 (86.8%)	0.552 (86.4%)	0.556 (87.0%)	**0.561 (87.9%)**	0.551 (86.3%)
Lucene	2.0-2.2	**0.714 (85.2%)**	0.694 (82.8%)	0.660 (78.7%)	0.698 (83.2%)	0.712 (84.9%)	0.679 (81.0%)	0.699 (83.4%)	0.694 (82.8%)	0.709 (84.6%)	0.710 (84.6%)	0.710 (84.7%)	0.710 (84.6%)
	2.2-2.4	0.627 (75.2%)	0.609 (73.0%)	0.616 (73.9%)	0.624 (74.9%)	**0.668 (80.2%)**	0.644 (77.2%)	0.620 (74.4%)	0.623 (74.7%)	0.642 (77.1%)	0.628 (75.3%)	0.631 (75.7%)	0.654 (78.4%)
Pbeans	1.0-2.0	0.709 (75.8%)	**0.820 (87.6%)**	0.710 (75.9%)	0.593 (63.4%)	0.781 (83.5%)	0.777 (83.1%)	0.776 (83.0%)	0.779 (83.3%)	0.756 (80.9%)	0.756 (80.8%)	0.767 (82.0%)	0.766 (81.9%)
Poi	1.5-2.0	0.626 (66.2%)	0.577 (61.1%)	0.597 (63.2%)	0.600 (63.5%)	0.700 (74.1%)	0.579 (61.3%)	0.675 (71.4%)	0.713 (75.5%)	0.689 (72.9%)	0.668 (70.7%)	0.646 (68.4%)	**0.718 (76.0%)**
	2.0-2.5	0.467 (61.8%)	0.473 (62.5%)	0.500 (66.1%)	0.503 (66.4%)	0.501 (66.3%)	0.493 (65.2%)	0.501 (66.3%)	0.508 (67.1%)	0.502 (66.4%)	**0.523 (69.1%)**	0.509 (67.3%)	0.498 (65.9%)
	2.5-3.0	0.659 (82.1%)	0.660 (82.3%)	0.646 (80.5%)	0.663 (82.6%)	**0.691 (86.1%)**	0.666 (83.0%)	0.645 (80.4%)	0.674 (84.0%)	0.678 (84.5%)	0.665 (82.9%)	0.682 (85.0%)	0.675 (84.1%)
Synapse	1.0-1.1	0.623 (68.5%)	0.608 (66.8%)	0.714 (78.5%)	0.682 (74.9%)	**0.741 (81.4%)**	0.692 (76.0%)	0.719 (79.0%)	0.731 (80.4%)	0.708 (77.8%)	0.707 (77.6%)	0.721 (79.2%)	0.735 (80.8%)
	1.1-1.2	0.652 (73.8%)	0.622 (70.3%)	0.679 (76.8%)	0.637 (72.0%)	0.663 (74.9%)	0.689 (77.9%)	**0.703 (79.5%)**	0.657 (74.3%)	0.678 (76.7%)	0.676 (76.5%)	0.696 (78.7%)	0.667 (75.5%)
Velocity	1.4-1.5	0.620 (77.7%)	0.632 (79.2%)	0.670 (83.9%)	0.636 (79.8%)	0.621 (77.8%)	0.641 (80.4%)	0.638 (80.0%)	0.646 (80.9%)	0.681 (85.3%)	0.650 (81.5%)	**0.690 (86.4%)**	0.650 (81.4%)
	1.5-1.6	0.760 (84.3%)	0.751 (83.4%)	0.685 (76.1%)	0.720 (79.9%)	0.757 (84.0%)	0.764 (84.8%)	**0.766 (85.1%)**	0.764 (84.9%)	0.758 (84.2%)	0.757 (84.1%)	0.753 (83.6%)	0.757 (84.1%)
Xalan	2.4-2.5	0.626 (77.2%)	0.619 (76.3%)	0.619 (76.3%)	**0.628 (77.5%)**	0.609 (75.1%)	0.622 (76.7%)	0.618 (76.2%)	0.621 (76.5%)	0.594 (73.2%)	0.603 (74.3%)	0.589 (72.6%)	0.598 (73.7%)
	2.5-2.6	0.647 (78.2%)	0.650 (78.6%)	0.628 (75.9%)	**0.670 (80.9%)**	0.650 (78.5%)	0.668 (80.8%)	0.652 (78.7%)	0.651 (78.7%)	0.655 (79.1%)	0.645 (78.0%)	0.665 (80.4%)	0.662 (80.0%)
Xerces	init-1.2	0.719 (74.9%)	0.703 (73.2%)	0.664 (69.2%)	0.669 (69.7%)	0.755 (78.6%)	0.703 (73.2%)	0.642 (66.9%)	0.660 (68.7%)	0.724 (75.4%)	**0.765 (79.7%)**	0.757 (78.8%)	0.756 (78.7%)
	1.2-1.3	0.561 (59.5%)	0.530 (56.2%)	0.546 (57.9%)	0.587 (62.3%)	0.524 (55.6%)	0.631 (67.0%)	0.534 (56.6%)	0.529 (56.1%)	0.729 (77.3%)	**0.745 (79.0%)**	0.729 (77.4%)	0.742 (78.7%)
Average		0.669 (75.0%)	0.672 (75.2%)	0.670 (75.1%)	0.670 (75.1%)	0.694 (77.8%)	0.699 (78.3%)	0.701 (78.5%)	0.697 (78.1%)	0.701 (78.5%)	0.709 (79.3%)	0.712 (79.7%)	**0.715 (79.9%)**

5.2 Deep Learning Based SDP methods

Since the advent of AlexNet [14] in 2012, deep learning has found widespread applications in various fields including software engineering. During recent years, many researchers have explored the use of deep learning in software defect prediction. Yang et al. [37] proposed a deep learning method for just-in-time defect prediction, which leveraged deep belief network (DBN) model to build a set of expressive features from the selected 14 basic change measures regarding code change. Li et al. [15] proposed a CNN-based defect prediction model, which used a CNN model to extract features from source code ASTs and combined them with 20 traditional hand-crafted features. Their results outperformed the DBN model [37] stated before and this is the very work that has provided the greatest inspiration for our current research. Two RNN-based studies [4], [34] employed LSTM and tree-base LSTM respectively to predict defects taking AST sequences as inputs. Pan et al. [23] used a deeper models on the basis of Li's CNN [15] to predict defects without employing any hand-crafted feature. These studies focused on a binary classification task rather than a ranking-oriented SDP problem. Some researchers also applied deep learning methods for solving ranking-oriented SDP problems, such as Lei et al.' work [27] and Meetesh and Pradeep's work [22]. However, they extracted features from traditional hand-crafted features instead of source code.

TABLE III
FPA AND RFPA (IN THE BRACKETS) RESULTS OF MODELS USING DIFFERENT FEATURE STRATEGIES, WITH RFR AS THE FINAL REGRESSOR.

Train Group		Hand-crafted Only	VGG-extracted Only	VGG Combined	GRU-extracted Only	GRU Combined	ResNet-extracted Only	ResNet Combined
Ant	1.3-1.4	**0.577 (63.8%)**	0.509 (56.2%)	0.543 (60.0%)	0.541 (59.7%)	0.549 (60.7%)	0.524 (57.9%)	0.564 (62.3%)
	1.4-1.5	**0.606 (63.3%)**	0.518 (54.2%)	0.512 (53.5%)	0.588 (61.5%)	0.574 (60.0%)	0.514 (53.7%)	0.584 (61.0%)
	1.5-1.6	0.725 (79.1%)	0.730 (79.6%)	0.737 (80.4%)	0.720 (78.5%)	0.738 (80.4%)	0.727 (79.2%)	**0.749 (81.7%)**
	1.6-1.7	0.805 (86.5%)	0.796 (85.6%)	0.798 (85.8%)	0.783 (84.1%)	0.794 (85.4%)	**0.812 (87.3%)**	0.800 (86.0%)
Camel	1.0-1.2	**0.658 (73.9%)**	0.590 (66.3%)	0.639 (71.9%)	0.564 (63.4%)	0.629 (70.7%)	0.569 (63.9%)	0.642 (72.2%)
	1.2-1.4	0.778 (81.9%)	0.800 (84.2%)	0.814 (85.6%)	0.819 (86.2%)	**0.827 (87.0%)**	0.785 (82.6%)	0.805 (84.7%)
	1.4-1.6	0.743 (78.8%)	0.772 (81.8%)	0.778 (82.5%)	0.783 (82.9%)	**0.794 (84.1%)**	0.789 (83.6%)	0.778 (82.5%)
Ivy	1.1-1.4	0.746 (76.8%)	0.772 (79.6%)	0.787 (81.0%)	0.837 (86.2%)	0.808 (83.3%)	**0.855 (88.1%)**	0.836 (86.1%)
	1.4-2.0	0.723 (75.7%)	**0.778 (81.4%)**	0.768 (80.3%)	0.772 (80.8%)	0.770 (80.6%)	0.698 (73.0%)	0.772 (80.7%)
JEdit	3.2-4.0	0.854 (90.6%)	0.859 (91.1%)	0.862 (91.4%)	0.862 (91.4%)	**0.867 (92.0%)**	0.845 (89.6%)	0.864 (91.6%)
	4.0-4.1	0.839 (89.9%)	0.830 (89.1%)	0.843 (90.4%)	0.833 (89.3%)	0.841 (90.2%)	0.831 (89.1%)	**0.855 (91.7%)**
	4.1-4.2	0.859 (89.3%)	0.867 (90.1%)	0.864 (89.8%)	0.859 (89.3%)	0.869 (90.4%)	0.868 (90.3%)	**0.873 (90.8%)**
	4.2-4.3	0.616 (62.2%)	0.743 (75.0%)	**0.773 (78.1%)**	0.781 (78.8%)	0.704 (71.1%)	0.725 (73.2%)	0.692 (69.8%)
Log4j	1.0-1.1	0.748 (83.8%)	0.777 (87.0%)	0.780 (87.4%)	0.782 (87.6%)	**0.787 (88.1%)**	0.775 (86.8%)	0.775 (86.8%)
	1.1-1.2	0.552 (86.4%)	0.549 (86.1%)	0.556 (87.0%)	0.557 (87.3%)	**0.561 (87.9%)**	0.552 (86.4%)	0.551 (86.3%)
Lucene	2.0-2.2	0.709 (84.6%)	0.710 (84.7%)	0.710 (84.6%)	**0.713 (85.0%)**	0.710 (84.7%)	0.700 (83.5%)	0.710 (84.6%)
	2.2-2.4	0.642 (77.1%)	0.613 (73.5%)	0.628 (75.3%)	0.624 (74.8%)	0.631 (75.7%)	**0.658 (78.9%)**	0.654 (78.4%)
Pbeans	1.0-2.0	0.756 (80.9%)	**0.785 (84.0%)**	0.756 (80.8%)	0.769 (82.2%)	0.767 (82.0%)	0.765 (81.8%)	0.766 (81.9%)
Poi	1.5-2.0	0.689 (72.9%)	0.682 (72.2%)	0.668 (70.7%)	0.664 (70.3%)	0.646 (68.4%)	0.713 (75.5%)	**0.718 (76.0%)**
	2.0-2.5	0.502 (66.4%)	0.525 (69.4%)	0.523 (69.1%)	0.528 (69.8%)	0.509 (67.3%)	**0.591 (78.2%)**	0.498 (65.9%)
	2.5-3.0	0.678 (84.5%)	0.664 (82.8%)	0.665 (82.9%)	0.669 (83.4%)	**0.682 (85.0%)**	0.660 (82.3%)	0.675 (84.1%)
Synapse	1.0-1.1	0.708 (77.8%)	0.651 (71.6%)	0.707 (77.6%)	0.650 (71.4%)	0.721 (79.2%)	0.615 (67.6%)	**0.735 (80.8%)**
	1.1-1.2	0.678 (76.7%)	0.669 (75.7%)	0.676 (76.5%)	0.680 (76.9%)	**0.696 (78.7%)**	0.678 (76.7%)	0.667 (75.5%)
Velocity	1.4-1.5	0.681 (85.3%)	0.643 (80.5%)	0.650 (81.5%)	0.621 (77.8%)	**0.690 (86.4%)**	0.671 (84.1%)	0.650 (81.4%)
	1.5-1.6	**0.758 (84.2%)**	0.754 (83.7%)	0.757 (84.1%)	0.741 (82.2%)	0.753 (83.6%)	0.744 (82.6%)	0.757 (84.1%)
Xalan	2.4-2.5	0.594 (73.2%)	0.609 (75.1%)	0.603 (74.3%)	0.600 (74.0%)	0.589 (72.6%)	**0.615 (75.8%)**	0.598 (73.7%)
	2.5-2.6	0.655 (79.1%)	0.651 (78.7%)	0.645 (78.0%)	0.654 (79.1%)	**0.665 (80.4%)**	0.647 (78.1%)	0.662 (80.0%)
Xerces	init-1.2	0.724 (75.4%)	0.638 (67.6%)	0.745 (79.0%)	0.538 (57.0%)	0.729 (77.4%)	**0.757 (78.8%)**	0.756 (78.7%)
	1.2-1.3	0.729 (77.3%)	0.756 (78.7%)	0.765 (79.7%)	0.750 (78.1%)	**0.757 (78.8%)**	0.537 (57.0%)	0.742 (78.7%)
Average		0.701 (78.5%)	0.698 (78.1%)	0.709 (79.3%)	0.699 (78.2%)	0.712 (79.7%)	0.697 (78.1%)	**0.715 (79.9%)**

6. THREATS TO VALIDITY

We implemented our DNN models using Pytorch and the predicted results might be diverse when implementing with other deep learning platforms like Tensorflow [1] or Keras [3]. The parameter selections for models are mostly based on trial-and-error and empirical knowledge without too much careful consideration. Hence the currently selected parameters may not be the most optimal for our models.

For this study, we focus on the ranking-oriented cross-version-WPDP task. There are within-version-WPDP and CPDP, which means that our approach has only been validated within a limited range and requires further validation on all kinds of SDP tasks. Additionally, we conducted our experiments only on the modified PSC dataset provided by [23], which is only a small part of all datasets.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a universal ranking-oriented software defect prediction framework called United Deep Network (UDN), which first extracts features from source code using deep learning for ranking-oriented SDP. UDN utilizes deep neural networks (DNN) like CNN and RNN for automated feature generation from source code with the syntactic and structural information preserved, and can combine DNN-learned features with discriminative traditional hand-crafted features.

We conduct experiments over 29 sets of open-source defect prediction data using the proposed feature combination strategy, and compare different combinations of backbone DNNs and regressors, as well as different feature strategies. The experimental results prove that the proposed feature combination strategy can significantly enhance the ranking performance of

nonlinear regression models like RFR, and all sorts of deep neural models like VGG, GRU and ResNet can benefit from this strategy.

In our future work, we will investigate the strategy on a wider range of deep neural network like transformer and attempt to explore the best combination of the DNN and regression model. We will also try to extend this strategy to cross-project defect prediction. Additionally, we will make attempts to investigate diverse approaches for utilizing hand-crafted features.

ACKNOWLEDGMENT

This work is supported by Higher Education Stability Support Program General Project (Grant No. 20220715114836001) . This work is also supported by National Natural Science Foundation of China (Grants No. 61602534) and the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001).

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation.

[3] François Chollet et al. Keras. https://keras.io, 2015.

[4] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. A deep tree-based model for software defect prediction.

[5] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. 17(4):531–577.

[6] Maurice H. Halstead. *Elements of software science*. Number 2 in Operating and programming systems series. North Holland, 3. print edition.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition.

[8] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite: a measurement theory perspective. 22(4):267–271.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. 9(8):1735–1780.

[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[11] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 414–423. ACM.

[12] Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall series in artificial intelligence. Prentice Hall.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105. Curran Associates Inc. event-place: Lake Tahoe, Nevada.

[15] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE.

[16] T.J. McCabe. A complexity measure. SE-2(4):308–320.

[17] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. 17(4):375–407.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space.

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality.

[20] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 382–391. IEEE.

[21] Meetesh Nevendra and Pradeep Singh. Defect count prediction via metric-based convolutional neural network. 33(22):15319–15344.

[22] Meetesh Nevendra and Pradeep Singh. Defect count prediction via metric-based convolutional neural network. *Neural Comput. Appl.*, 33(22):15319–15344, nov 2021.

[23] Cong Pan, Minyan Lu, Biao Xu, and Houleng Gao. An improved CNN model for within-project software defect prediction. 9(10):2138.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chin-

tala. PyTorch: An imperative style, high-performance deep learning library.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[26] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In Songmao Zhang, Martin Wirsing, and Zili Zhang, editors, *Knowledge Science, Engineering and Management*, volume 9403, pages 547–553. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[27] Lei Qiao, Xuesong Li, Qasim Umer, and Ping Guo. Deep learning based software defect prediction. 385:100–110.

[28] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition.

[30] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 99–108. IEEE.

[31] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM.

[32] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Comparing the effectiveness of several modeling methods for fault prediction. 15(3):277–295.

[33] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE.

[34] Kerong Ben Xian Zhang and Jie Zeng. Using cross-entropy value of code for better defect prediction. 14(9):2105. Publisher: Int J Performability Eng.

[35] Xiaoxing Yang, Ke Tang, and Xin Yao. A learning-to-rank approach to software defect prediction. 64(1):234–246.

[36] Xiaoxing Yang and Wushao Wen. Ridge and lasso regression models for cross-version defect prediction. 67(3):885–896.

[37] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE.

[38] Xiao Yu, Kwabena Ebo Bennin, Jin Liu, Jacky Wai Keung, Xiaofei Yin, and Zhou Xu. An empirical study of learning to rank techniques for effort-aware defect prediction. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 298–309. IEEE.

[39] Xiao Yu, Heng Dai, Li Li, Xiaodong Gu, Jacky Wai Keung, Kwabena Ebo Bennin, Fuyang Li, and Jin Liu. Finding the best learning to rank algorithms for effort-aware defect prediction. 157:107165, 2023.

[40] Manzil Zaheer, Sashank J. Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive methods for nonconvex optimization. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 9815–9825, Red Hook, NY, USA, 2018. Curran Associates Inc.

[41] Zuhaira Muhammad Zain, Sapiah Sakri, and Nurul Halimatul Asmak Ismail. Application of deep learning in software defect prediction: Systematic literature review and meta-analysis. 158:107175.

[42] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning.

[43] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9. IEEE.