APPLICATION OF XGBOOST ALGORITHM AND DEEP LEARNING TECHNIQUES FOR SEVERITY ASSESSMENT OF SOFTWARE DEFECT REPORTS

Ruchika Malhotra
Department of Computer Science and Engineering, Delhi Technological University, Delhi, India
ruchikamalhotra2004@yahoo.com

Akanksha Chauhan
Department of Computer Science and Engineering, Delhi Technological University, Delhi, India
akankshac36@gmail.com

Abstract - Software is present in every aspect of our everyday life, and defects are bound to be found during the testing of the software, no matter how small. It is therefore imperative for software testing engineers to assess the severity of software defects to allocate proper resources for the correction of the defects and prevent software crashes. In this paper, we have proposed the use of the Extreme Gradient Boosting Technique (XGBoost) and deep learning techniques: CNN (Convolutional Neural Network) and RNN (Recurrent Neural Network) to predict the severity of the defects occurring in the software. AUC and sensitivity are the metrics used to evaluate the results. All three techniques: XGBoost algorithm, CNN and RNN have performed really well in predicting the severities for all the defects. It has also been noted that XGBoost algorithm is the most efficient in predicting high severity defects, while the performance of deep learning techniques is excellent for the highest as well as the lowest severity defects. For the rest of the severity values, the performance of all the three classifiers is fairly consistent.

***Keywords*:** deep learning; software defects; severity; severity assessment.

1. Introduction

The severity of a defect may be defined as the impact of failure on a software. In critical systems, mainly real-time systems like those of NASA, it is highly imperative that the testing engineers are sure that the software will not crash while operating as it may cause major damage to the project as well as the reputation of the organization.

Software is bound to pick up some defects while in the development phase, and it is the job of the testing engineers to assess these defects and their impact on the software. Note that the severity and priority of a defect may not be the same. For example, it may be the case that a defect is required to be removed immediately as it is not letting the users/customers to proceed further but the damage caused by the defect is not that high. In such cases, the priority of the defect is high, but the severity is not. Here, however, we are discussing the severity of the defect, i.e., the defect may not occur immediately, but if and when it occurs, it may cause some severe damage.

In software testing, it is a well-known fact that the sooner the defects are recognized, the lesser it costs to correct them, hence, minimizing the overall cost of software development. Here, we propose an automated method to assess the severity of the software defects by using machine learning and deep learning techniques. Software defect reports are generated by the user of the software whenever the software does not perform in its intended manner. Software such as Jira and Bugzilla are used to report the defect. These defect reports are in the form of text, which is an example of highly unstructured data. Though a lot of work has been done in this field, there's still a long way to go when it comes to the use of ensemble methods and deep learning techniques to predict the severity. In this paper, text mining techniques have been employed along with XGBoost and deep learning techniques for predicting the severity of software defect reports. We have used XGBoost, which is the strongest ensemble machine learning method at the time of writing of this paper. It is trusted by a number of winning teams of machine learning competitions. The performance of this method is comparable to deep learning methods. We have also employed deep learning techniques: CNN and RNN using word embeddings for the same.

The paper is organized as follows: Section 2 analyses the work that has been done till date in the same field. Section 3 covers the methodology used in the proposed approach. Section 4 is the case study which covers details of the dataset, process, metrics used and evaluation of results. Section 5 sums up the paper.

2. Related Work

Menzies and Marcus [1] were the first ones who worked on the severity assessment of software defect reports and used rule learning algorithms. They designed a tool named SEVERIS (SEVERity ISsue assessment). SEVERIS was applied to NASA's PITS (Project and Issue Tracking System) database. PITS contained data collected over ten years including issues of robotic satellite mission and human-rated systems. The system was applied to 5 datasets from PITS database: {pitsA, pitsB, pitsC, pitsD, pitsE} which contained issues of five robotic missions of NASA. The tool was used to review reports and alert if the predicted severity was anomalous.

Cubranic and Murphy [2] used machine learning techniques for automatic bug triage. Bug triage refers to what needs to be done with bug as it is reported. Their work was to assign developers who should be working on a particular bug based on the description of the bug as entered by the user and developers' skills. They tested their approach on Eclipse bug reports and used Naive Bayes Classifier for classifying the bug reports.

Lamkanfi et al. [3] compared various machine learning methods to assess the severity of reported bugs on two open-source systems: Eclipse and GNOME. They clubbed the six severity levels: blocker, critical, major, normal, minor, and trivial into two: severe and non-severe. Non-severe included trivial and minor whereas severe included blocker, critical and major. Normal severity bugs were not taken into account. The classifiers used by Lamkanfi et al. [3] were Naive Bayes, 1-Nearest neighbor, Naive Bayes Multinomial, and SVM using RBF kernel.

Orthogonal Defect Classification (ODC) is the most influential framework for defect classification and analysis. However, it does require intensive human labor and expertise of both ODC and domain knowledge to classify the defects. Huang et al. [6] worked to automate ODC by treating it as a supervised text classification problem by using SVM classifier.

Patil used Explicit Semantic Analysis to compute semantic similarity between defect reports and defect labels based on the concept. The defect label was assigned to the defect report based on its similarity with the defect report[7].

Yang et al. [8] worked with three prevalent feature selection schemes: information gain, correlation coefficient, and chi-square to select the best features and finally used Multinomial Naive Bayes Classifier to predict the bugs.

Yang et al. [9] worked on a textual emotion words-based dictionary, combining it with Naive Bayes Multinomial classifier to assess the severity of defect reports.

Chaturvedi and Singh in [11] determined the bug severity on the software bug reports dataset obtained from NASA's PROMISE repository. They used the following machine learning techniques: Naïve Bayes, Support Vector Machine, Naïve Bayes Multinomial, k-Nearest Neighbour, J48, and RIPPER to predict the severity of bugs in the bug reports.

Ramay et al. [12] applied deep learning methods to predict the severity of bug reports of two open-source systems: Eclipse and Mozilla. Similar to Lamkanfi et al. [3], they clubbed the six severities into two: severe and non-severe. They also considered the emotion score of the bug reports as users are very expressive about reporting the bugs. Senti4SD repository was used to calculate the emotion score of the bug reports. MNB, RF, CNN, and LSTM were used to predict the severity of the reported bugs.

In this paper, our goal is the same as Menzies and Marcus [1], that is to predict the severity of bug reports and we have employed XGBoost and deep learning methods: CNN and RNN.

3. Methodology Used

In this paper, we have worked on the dataset obtained from NASA's PITS database. The dataset contains Defect Reports in text form along with the severity of the defect. We divide the dataset in a ratio of 70% to train the machine learning and deep learning models and 30% to test the models.

Since the text is highly unstructured, and there are no predefined features present as those in structured data, we employ text mining techniques to extract the features from the data and reduce its dimension. After that, XGBoost (machine learning technique) and deep learning techniques are applied to assess the defect severity.

3.1 Tokenization

Tokenization is the process of converting a text into tokens. These tokens can be sentences, words, or characters. For our paper, we have tokenized the text up to word level. Word Tokenizer available in NLTK library is used to tokenize the text into words. Cleaning of text is also performed wherein all the punctuation marks, special symbols, numbers and unnecessary spaces are removed from the text. Therefore, a sentence ["Hey! look, what's there?"] gets converted to a list of words as [Hey look what s there].

3.2 Stop Word Removal

Stop words are the most regularly occurring words in any language. These may be prepositions, conjunctions or interjections which are used very often in the text and do not really add to the meaning of the text. Words like a, an, the, this, that, and, in, it, etc. are omitted from the text.

Stop words of the English language are available in NLTK library and can be imported from there. Stop words may also be specific to a particular application. For example, if there is an application related to the healthcare industry, the word 'doctor' might appear quite a lot of times and may not really add to the meaning of the text. In such cases, these additional words may be appended to the stop words list and all of them can be removed from the text in one go.

3.3 Stemming

Stemming and Lemmatization are performed to get the base/root form of each word. While both the techniques are used to get the base form of each word in the text, there is a major difference in how it is achieved in both the techniques.

Stemming works by simply stripping off of any suffixes or prefixes that might be present with the base word. By using stemming 'run', 'running', 'runner' gets converted to 'run'. But there might occur a case of overstemming and understemming. Often it might be the case that the words obtained after stemming may not make any sense because of the wrong context or wrong spelling.

Lemmatization converts each word to its base form by checking the lexicon, i.e., we can say that the root words obtained after lemmatization are morphologically correct. By using lemmatization, ‘caring' gets converted to ‘care’ which would have been converted to ‘car’ using stemming. Lemmatization, therefore, takes more time than stemming. Here in this paper, we have applied Stemming by using the PorterStemmer, as is the case in Menzies and Marcus [1].

*3.4 Tf*IDF*

Tf*IDF is the product of Term Frequency and Inverse Document Frequency. It assigns weight to each term in the text which signifies the importance of the term. The weight of each term is directly proportional to the number of occurrences of the term in a document while it is inversely proportional to the number of occurrences of the term in the corpus. It is used to normalize the weights of each term in cases where the occurrence of some terms is benefitted by the length of document.

Mathematically,

$$Tf * IDF = \frac{term}{totalTerms} \times log \frac{documents}{document[term]} \quad (1)$$

where, term is the term in consideration,

totalTerms is the total number of terms in a document, document[term] is the document containing term,

documents is the total number of documents.

For information retrieval and text mining applications, we select top k terms as ranked by the Tf*IDF score. These k terms then act as the features in our data. Rest of the features are not taken into further consideration. In this paper, we have considered top 100 features.

3.5 InfoGain

Information gain is basically the calculation of entropy or surprise element in a dataset. If a dataset is split in a particular ratio InfoGain measures the entropy introduced in the dataset before and after the split. Information gain is low for high frequency terms and high for rare terms.

We have used MutualInfoClassifier available in scikit-learn, which gives the mutual correlation between a term and the outcome. It employs information gain in the background and tells us how much impact a term has in a particular result. MutualInfo tells us how much information can be gained from a random variable.

3.6 XGBoost

Developed by Tianqi Chen and Carlos Guestrin at the University of Washington in 2014 [13], Extreme Gradient Boost or XGBoost is a scalable decision tree-based ensemble machine learning algorithm that uses a gradient boosting framework.

Ever since its introduction, XGBoost has proved to be the fastest of all the machine learning algorithms. In fact, it has a good competition with deep learning methods in terms of accuracy and score. Many teams that have won in various competitions organized by machine learning competition site Kaggle employed XGBoost. For unstructured data, neural networks still prove to be the most useful, but for small structured/tabular data, decision tree-based algorithm outperform all the other algorithms. XGBoost has given state-of-the-art results in many problems. It is for this reason that we have used this algorithm along with deep learning methods to predict the severity of software defect reports.

The architecture of the XGBoost algorithm is shown in the following figure. The process of boosting involves building strong classifiers from multiple weak classifiers iteratively. All samples in the dataset are assigned the same weights initially. The first weak classifier is trained by picking some of the samples from the dataset randomly. Every sample present in the dataset has an equal probability of being selected to be included in the training set.

Each weak classifier tests all the samples and then updates the weights for all the misclassified samples. These samples with their updated weights are used for the training of the next weak classifier. These weak classifiers work in a consecutive manner. While predicting the results for a test sample, predictions made by all the classifiers is taken into consideration and the majority of these predictions is the final prediction.

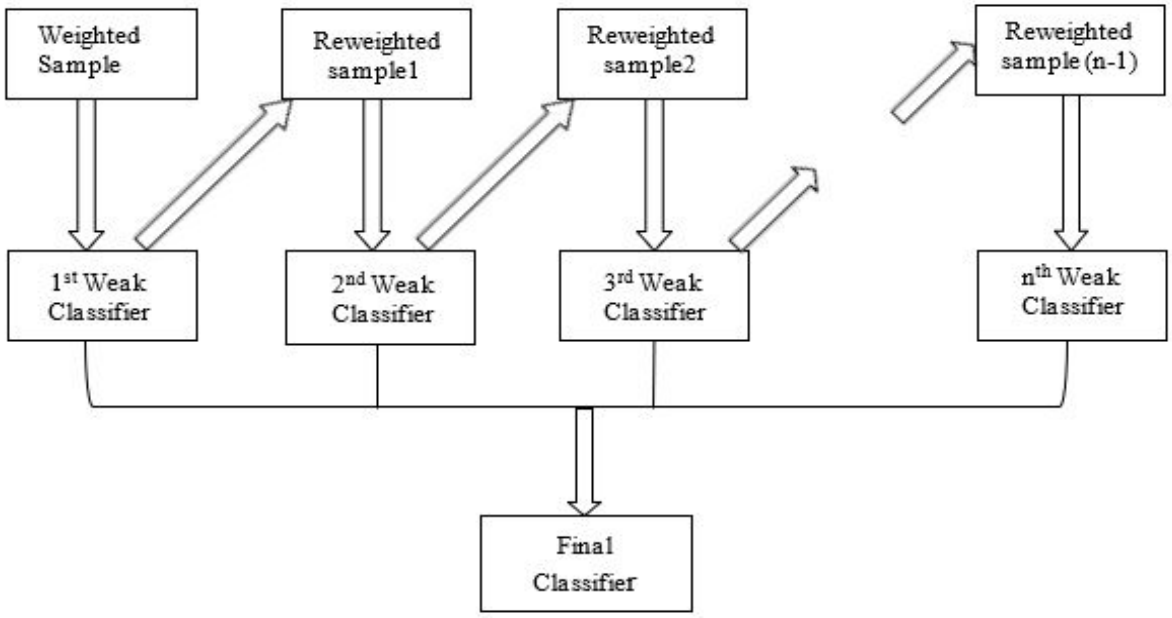

Figure 1. The architecture of XGBoost Algorithm

Regularization issues are not taken very seriously by most of the boosting algorithms. In Gradient Boosting Technique, some regularization parameters are present like maximum depth learning rate, minimum samples per leaf, etc. which can be used for controlling the tree structure. It is further improved by extreme gradient boosting. Therefore, extreme gradient boosting or XGBoost is a more regularized version of Gradient Boosted Trees.

Decision Tree → Bagging → Random Forest →Boosting →Gradient Boosting → XGBoost

Figure 2. Evolution of tree-based algorithms

The robustness of the model can be increased by changing the learning rate which minimizes weights on each step. We have taken 0.1 as the learning rate. The maximum depth of the tree is set to 6 to avoid overfitting of the model. The number of trees generated is given by the parameter n estimators and its value is set to 100. The objective function used is the multi-class version of the softmax function.

3.7 Word Embeddings

To feed the text into the neural network, we transform the text into a vector representation. These vector representations for text are termed as word embeddings. Word embeddings are considered a major breakthrough when it comes to Natural Language Processing using Deep Learning.

Each word is represented using a real-valued word vector often having tens or hundreds of dimensions. The vector values are learned in a way that resembles a neural network, and hence the technique is used in deep learning. Words having similar meanings have similar representations. The word embedding algorithm used in this case study was Global Vectors for Word Representation (GloVe) [14], which is an extension to the word2vec method for learning word vectors. GloVe combines the global statistics of matrix factorization techniques like LSA with local context-based learning in word2vec. Each word is represented by the vector closest to the point obtained corresponding to that word. Similar words are closer to each other.

3.8 Deep Learning Techniques used

3.8.1 CNN

Convolutional Neural Networks are several layers of convolutions. A convolution may be defined as a sliding window function applied to a matrix. These functions are non-linear activation functions like relu or tanh. The convolutions are used over the input layer to calculate the output. Each layer may use a different convolution.

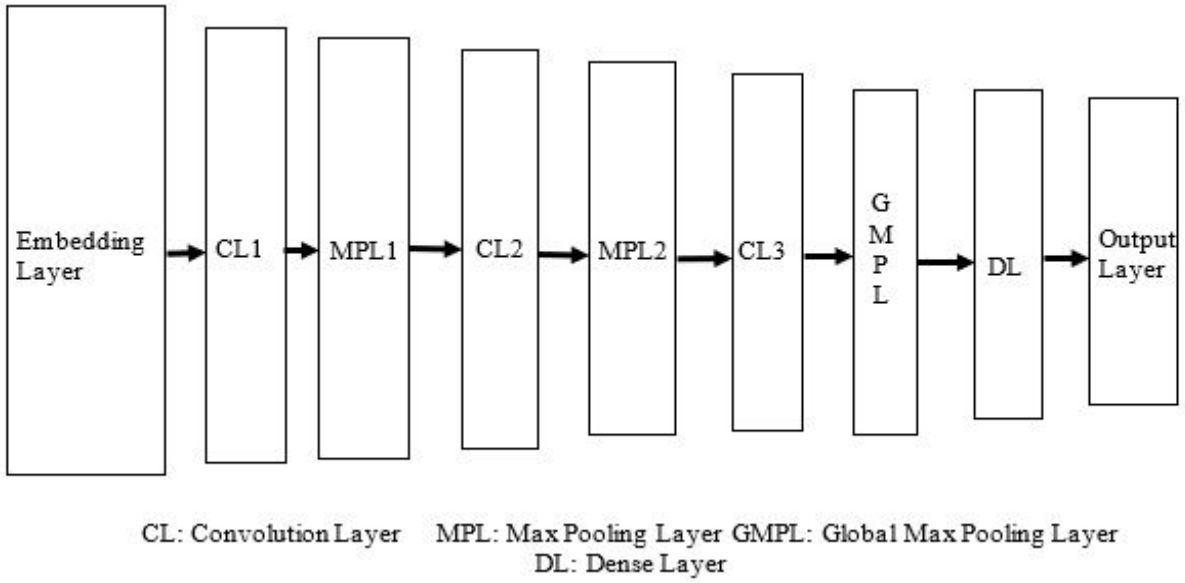

Figure 3. The architecture of CNN

A pooling layer is present after each convolution layer, which subsamples the output of the previous convolution layer and feeds it as an input to the next layer.

The simplest pooling operation may be a max operation applied to the result of each filter. A global max pooling is performed and its result is fed to the final fully-connected/ dense layer. Finally, the classification is performed by a softmax classifier. We use three 1D convolution layers with relu as the activation function, which decides the output. We have chosen the filter value to be 128 which indicates the number of neurons, and the value of kernel size is set to 3, which indicates the size of the filter. Global max pooling is performed, and the output of the global max-pooling layer serves as the input to a dense layer having 128 neurons.

Output layer maps input to a single output, which is the predicted severity of the corresponding bug report. We have used categorical_crossentropy as the loss function for the model, which is a cross-entropy loss that is used to measure the performance of a multi-class classification model.

3.8.2 RNN

Traditional neural networks lacked persistence. They could not make an informed decision about an event based on the previous occurrences of that event. In simpler terms, they lacked memory.

Recurrent Neural Networks were designed to overcome this issue. RNNs have loops in them, which allow the persistence of information. Traditional RNNs are able to connect previous relevant information to the present task, but when the gap between the relevant information and the current task becomes large, they fail, *viz.* they are unable to handle long term dependencies. To address this issue, a special kind of RNN was developed known as Long Short-Term Memory (LSTM). LSTMs are capable of learning long term dependencies. It is in their innate nature to remember information for long periods of time. The repeating module in a LSTM has four neural network layers, which interact in a unique way as compared to the standard RNN that has just one neural network layer in its repeating module. The key components of LSTM are a cell state and three gates. The LSTM can add or remove information from the cell state with the help of gates.

The remarkable results that people have achieved using RNNs are due to the LSTM only. They are highly effective for text classification problems and are also strongly recommended for the same.

In our paper, we feed the input to the RNN model using the embedding layer, which consists of the embedding matrix. The input goes through an LSTM network. The output of the LSTM network is passed through a global max-pooling layer. The output of the global max-pooling layer is passed onto a fully connected dense layer having 50 neurons and finally, the output layer declares its verdict on the classification of the defect report into one of the five available classes by using the softmax activation function.

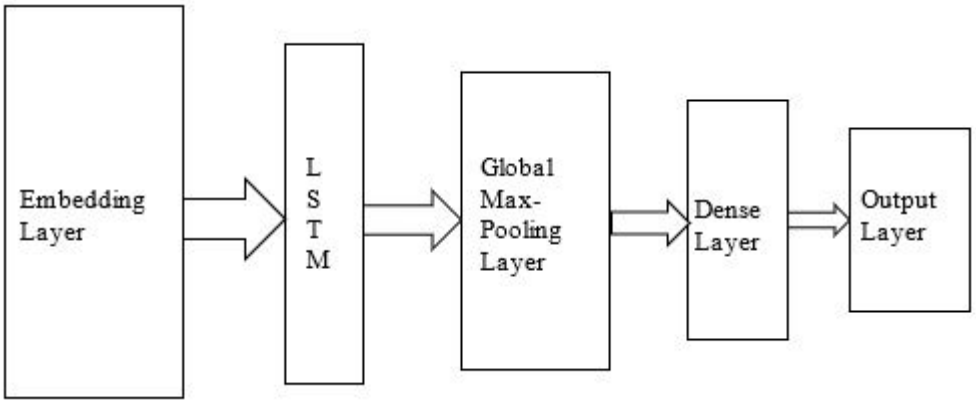

Figure 4. Architecture of RNN

4. Case Study

4.1 Dataset

We evaluated our system on NASA's original pitsA dataset. The dataset is available at https://zenodo.org/ . It is to be noted that there are five levels of severity, labeled 1 through 5. Label 1 is the most critical of all the bugs and may prove to be fatal to the project as well as humans involved in the project. Label 1 bugs are therefore very less since every possible measure is taken in order to avoid them. In the pitsA dataset, label 1 bugs are not present. Severity decreases with the rise in the number of labels, Label 5 being the most trivial of all the bugs. Label 5 is so trivial that more often than not the bug is corrected without even reporting. Therefore, there are very less reports having label as 5 which makes the data highly imbalanced.

We have used two attributes of the dataset: description attribute gives the bug report and severity attribute that gives the priority of the resolution of the bug. The total number of bug reports in the dataset is 965. Label 1 bugs are not present in this dataset, label 2 bugs are 325 in number, label 3 bugs are 375, label 4 bugs are 239 and there are 26 label 5 bugs.

4.2 Process

Cleaning of data is performed as described in section 3 by removing all the punctuation marks and special symbols. Stop words removal is performed, stemming is done. To proceed with XGBoost, tf-idf vectoriser is used for feature extraction and then information gain is used for feature selection. Finally, the XGBoost algorithm is applied on the data.

We have used two deep learning techniques: CNN and RNN in this paper to automatically learn and classify defect reports into one of the 5 categories in our test dataset. Preprocessing of data is performed as given in section 3. We tokenize the data, remove stop words and perform stemming. GloVe word embeddings are used to convert this data into a word-level matrix where each word is represented by a vector. Then, before feeding any input to the model, we create an embedding layer. After the text is converted into a word-level matrix using word embeddings: GloVe having a dimension of 300, we use this weighted matrix as an input to the embedding layer. Both CNNs and RNNs require input to have a static size and sentence lengths can vary greatly. Therefore, we chose a maximum sentence length of 200, i.e., a sentence can have a maximum of 200 words only. If a sentence contained less than 200 tokens, a special stop word was repeatedly appended to the start of the sentence to meet the 200-word requirement. If a sentence contained over 200 words, only the first 200 were considered to be representative of that sentence. The CNN and RNN models are finally fed with the output of this layer.

The dataset is split into training and testing sets in the ratio of 70:30. In each run, there are different training and testing sets based on a partitioning variable (random state). The validation is performed on ten different values of partitioning variable to get more accurate and generalized results.

4.3 Metrics

We have used sensitivity, AUC measure, and accuracy as our metrics, all of which are well-known metrics.

- **Sensitivity**

While running the predictions on our dataset, the results can be broken down into four parts:

True Positives: Reports that have defects and are predicted correctly.

False Positives: Reports that do not have defects and are predicted to have defects.

True Negatives: Reports that are defect-free and are predicted correctly.

False Negatives: Reports that have defects and are predicted as defect-free.

Also called recall, sensitivity is the metric that gives the model's ability to predict true positives of each category, i.e., it provides the percentage of reports that have defects and are correctly predicted so. Mathematically,

$$Sensitivity = \frac{TP}{(TP+FN)} \times 100. \quad (2)$$

- **AUC measure**

Receiver Operating Characteristics (ROC) is a probability curve that is created by plotting the True Positive Rate (TPR)/Sensitivity against the False Positive Rate (FPR). AUC is the area under the ROC curve and represents the degree of separability. It indicates how capable the model is, in discriminating between the classes. The value of AUC lies between 0 and 1. Higher values of AUC represent the model's ability to distinguish between the classes more efficiently. In the multi-class model, we can plot multiple ROC curves corresponding to each class using one versus all approaches.

- **Accuracy**

Accuracy describes the closeness of a measurement to the true value. The accuracy is defined as the average number of correct predictions in the case of multi-class classification.

Metrics calculation is done for each individual run and then overall metric calculation is done using the macro averaging. As the PITS dataset is imbalanced, we take one versus one macro average in the case of AUC as it is insensitive to class imbalance and gives a better picture of the model.

4.4 Results

We have divided the whole dataset into training and testing sets in a ratio of 70:30 based on different partitioning variables. For each partitioning variable, a unique set is generated corresponding to that variable and in each set, 70 percent of the data will be used as the training set and 30 percent will be used as the testing set. These same training sets will then be used by the XGBoost, CNN and RNN models to train, and the corresponding testing sets will be used by these models to predict the severity of defect reports.

We have performed the validation on ten different values of the partitioning variable for XGBoost and five different values each for CNN and RNN to get more accurate and generalized results.

4.4.1 XGBoost

The following Table 1 shows the individual results for each severity for various runs of XGBoost and whereas the overall results of the model obtained by taking a macro average of the individual results are shown in Table 2. The results have been evaluated corresponding to the top 100 features with AUC, sensitivity, and accuracy as the evaluation metrics.

Table 1. Individual Results of XGBoost

S. No.	Sev 2		Sev 3		Sev 4		Sev 5	
	AUC	Sensitivity	AUC	Sensitivity	AUC	Sensitivity	AUC	Sensitivity
1	0.88	81.44	0.79	81.03	0.78	64.79	0.75	50
2	0.87	77.78	**0.83**	**85.84**	0.81	70	0.81	62.50
3	0.88	81.44	0.82	85.83	0.74	55.56	0.80	60
4	0.90	85.10	0.78	78.45	0.78	65.28	0.75	50
5	0.88	82.24	0.76	76.19	0.77	61.76	0.65	30
6	**0.91**	**86.36**	0.78	78.76	0.78	65.82	0.75	50
7	0.85	77.57	0.80	79.63	**0.81**	**70.77**	0.80	60
8	0.88	80.95	0.76	81.03	0.72	52.44	0.81	62.50
9	0.90	86.11	0.80	81.73	0.77	60.87	0.78	55.56
10	0.90	84.91	0.78	78.43	0.76	61.11	**0.85**	**70**

Table 2. Overall Results of XGBoost

S. No.	AUC	Sensitivity	Accuracy
1	0.795	69	76.55
2	**0.82**	**74.02**	**78.62**
3	0.80	70.71	76.9
4	0.80	69.71	76.55
5	0.75	62.55	73.45
6	0.80	70.24	76.55
7	0.81	71.99	76.21
8	0.79	69.23	72.41
9	0.81	71.07	77.59
10	0.82	73.61	76.21

As we can see from the above Table 1 that the AUC measure of the model comes out to be 0.91 for severity 2 and the corresponding sensitivity comes out to be 86.36%. Whereas, as the severity is decreasing (increasing number), the AUC measure declines to 0.83 for severity 3, corresponding sensitivity being 85.84%. For severity 4, these values further decrease to AUC being 0.81 and sensitivity being 70.77%. However, for the severity value 5, the AUC measure shows a slight increase, and the value comes out to be 0.85 while the sensitivity is 70%. The trend in these values shows that the model is the most efficient in predicting high severity (severity 2) values, though, other severities do not lag behind by much. Considering AUC and sensitivity as the measure for performance evaluation, we can say that the model is consistent with respect to all the severity levels.

From Table 2 we can see the overall results. The overall AUC which is obtained by the one versus one macro averaging comes out to be 0.82 while sensitivity comes out to be 74.02% with an overall accuracy of 78.62%.

4.4.2 CNN

The following Table 3 shows the individual results for each severity for various runs of CNN, and the overall results of the model obtained by taking the macro average of the metrics used are shown in Table 4. The results have been evaluated corresponding to the top 200 features and the evaluation metrics used are AUC, sensitivity, and accuracy.

Table 3. Individual Results of CNN

S.No.	Sev 2		Sev 3		Sev 4		Sev 5	
	AUC	Sensitivity	AUC	Sensitivity	AUC	Sensitiv-	AUC	Sensitivity
1	**0.95**	**87.23**	**0.89**	87.96	**0.91**	59.26	0.95	57.14
2	0.92	72.92	0.89	70.18	0.88	**84.51**	**0.96**	**72.78**
3	0.95	72.92	0.88	**92.97**	0.88	44.64	0.95	60
4	0.93	79.79	0.85	80.91	0.86	62.16	0.96	66.67
5	0.95	83.67	0.87	84.29	0.87	53.97	0.94	62.5

Unlike XGBoost, we can see from the above Table 3 that the highest value of AUC comes out to be 0.96 for severity 5 with corresponding sensitivity being 72.78%. The second highest value is obtained for severity 2 with a sensitivity of 87.23%. For severity 3, the AUC comes out to be 0.89, and the highest sensitivity obtained is 92.97%. For severity 4, the AUC obtained is 0.91, and the highest sensitivity obtained is 84.51%. As is clear from the above table, considering AUC and sensitivity as the evaluation measure, we can say that the performance of the model is consistent and the model is efficient for all the severities.

Table 4. Overall Results of CNN

S.No.	AUC	Sensitivity	Accuracy
1	**0.92**	72.90	**79**
2	0.91	**76.34**	75
3	0.91	67.63	76
4	0.90	72.38	75
5	0.91	71.11	77

The Table 4 shows the overall results of CNN over five runs of the partitioning variable. The overall results are calculated in a manner similar to that of XGBoost by taking the macro average of the individual results. The overall value of AUC is evaluated using one-versus-one macro averaging, which is insensitive to the class imbalance present in the data. The highest value of overall AUC comes out to be 0.92, while sensitivity and accuracy come out to be 76.34% and 79%, respectively. The high value of AUC suggests that the model is capable of differentiating between the classes.

4.4.3 RNN

Similar to XGBoost and CNN, RNN model performed various runs for different values of the partitioning variable. Individual results of each severity are being shown in Table 5. The results have been evaluated corresponding to the top 200 features, and the evaluation metrics used are AUC, sensitivity, and accuracy.

Table 5. Individual Results of RNN

S.No.	Sev 2		Sev 3		Sev 4		Sev 5	
	AUC	Sensitivity	AUC	Sensitivity	AUC	Sensitivity	AUC	Sensitivity
1	0.93	77.08	0.87	**87.72**	0.88	57.75	0.95	66.66
2	**0.96**	**85.42**	**0.88**	85.16	0.90	55.36	**0.96**	50
3	0.94	79.57	0.87	79.84	**0.91**	**72.73**	0.96	**71.43**
4	0.95	82.24	0.87	74.76	0.90	71.83	0.96	55.56
5	0.96	83.33	0.87	78.07	0.86	61.97	0.94	55.56

From the individual results for each severity (Table 5) we can deduce that the AUC value for both severity 2 and 5 comes out to be 0.96 with their respective sensitivities being 85.42% and 71.43%. For severity 3, AUC value is 0.88, and the sensitivity is 87.72%. For severity 4, AUC value is 0.91, and the sensitivity is 72.73%. Considering AUC and sensitivity as the evaluation measure, we can see that the performance of the model is consistent. The value of AUC being close to 1 suggests the model's capability to differentiate between the classes.

Table 6. Overall Results of RNN

S.No.	AUC	Sensitivity	Accuracy
1	0.91	72.3	76
2	0.91	68.98	78
3	**0.91**	**75.89**	**78**
4	0.91	71.09	76
5	0.89	69.73	75

Table 6 shows the overall results of the various runs of RNN obtained by macro averaging the individual results. The highest value of overall AUC comes out to be 0.91 while the highest sensitivity is 75.89 and the highest accuracy is 78. The value of AUC being so close to 1 suggests the model's capability to differentiate between the classes.

5. Conclusion

Softwares are present in every aspect of our lives, and it is quite inevitable for these softwares to have defects. Users report these defects with severity, and it helps developers to work on these defects to correct them in a priority wise manner. Our proposed approach employs XGBoost and deep learning techniques to predict the severity of defect reports. We have used the PITS A dataset available in NASA's PITS database to evaluate the model. The results were analysed using the AUC measure and sensitivity as the metrics. We concluded that with XGBoost, the model performs well in predicting the high severity (Severity 2) defects. While with both the deep learning techniques, CNN, and RNN, the models were exceptionally well in predicting the highest as well as the lowest priority of the defect reports (severity 2 and severity 5), and for the rest of the severities, the models' performances were fairly consistent.

As of now, we have based our work on PITS A dataset only and we aim to further extend it to other PITS datasets as well as other datasets.

References

[1] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports", 2008 IEEE International Conference on Software Maintenance, Beijing, 2008, pp. 346-355.
[2] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization", Proceedings of the International Conference on Software Engineering Knowledge Engineering, Alberta, 2004, pp.92-97.
[3] A. Lamkanfi, J. Péérez, S. Demeyer, "The eclipse and mozilla defect tracking dataset: A genuine dataset for mining bug information", Proc. 10th Work. Conf. Mining Softw. Repositories (MSR), May 2013, pp. 203-206.
[4] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, "Predicting the severity of a reported bug", Proc. 7th IEEE Working Conf. Mining Softw. Repositories (MSR), May 2010, pp. 1-10.
[5] A. Lamkanfi, S. Demeyer, Q. D. Soetens, T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug", Proc. 15th Eur. Conf. Softw. Maintenance Reeng., Mar. 2011, pp. 249-258.
[6] L. Huang, V. Ng, I. Persing et al., "AutoODC: Automated generation of orthogonal defect classifications", Autom Softw Eng 22, 2015, pp. 3-46.
[7] S. Patil, "Concept-Based Classification of Software Defect Reports," 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), Buenos Aires, 2017, pp. 182-186.
[8] C. Z. Yang, C. C. Hou, W. C. Kao, X Chen, "An empirical study on improving severity prediction of defect reports using feature selection", Proc. 19th Asia-Pacific Software Engineering Conference, 2012, pp. 240-249.
[9] G. Yang, S. Baek, J.-W. Lee, B. Lee, "Analyzing emotion words to predict severity of software bugs: A case study of open source projects", Proc. Symp. Appl. Comput., Apr. 2017, pp. 1280-1287.
[10] G. Yang, T. Zhang, B. Lee, "An emotion similarity based severity prediction of software bugs: A case study of open source projects", IEICE Trans. Inf. Syst., vol. E101.D, 2018, pp. 2015-2026.
[11] K. K. Chaturvedi, V. B. Singh, "Determining bug severity using machine learning techniques", Proc. CSI 6th Int. Conf. Softw. Eng. (CONSEG), Sep. 2012, pp. 1-6.
[12] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu and I. Illahi, "Deep Neural Network-Based Severity Prediction of Bug Reports", in IEEE Access, vol. 7, 2019, pp. 46846-46857.
[13] T. Chen, C. Guestrin, "XGBoost: A Scalable Tree Boosting System", Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 785-794.
[14] J. Pennington, R. Socher, C. Manning, "Glove: Global Vectors for Word Representation", EMNLP. 14, 2014, pp. 1532-1543.
[15] R. Jindal, R. Malhotra, A. Jain, "Analysis of Software Project Reports for Defect Prediction Using KNN", Lecture Notes in Engineering and Computer Science, vol. 2211, no. 1, Jul. 2014, pp. 180–185.
[16] R. Jindal, R. Malhotra, A. Jain, "Software defect prediction using neural networks", Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization, Noida, 2014, pp. 1-6.
[17] R. Malhotra, N. Kapoor, R. Jain, S. Biyani, "Severity Assessment of Software Defect Reports using Text Classification", International Journal of Computer Applications, 83, 2013, pp. 13-16.