Predicting Software Bugs using Deep Learning: A Comprehensive Review

Avinash Kori[1], Joy Bhattacharji[2], Prof. Anshul Jain[3]
[1]M. Tech Scholar, [2, 3]Assistant Professor
[1, 2, 3]Department of CSE, [1,2]TIT-A, [3]TIT, Bhopal, India

Abstract: Defect prediction is one of the key challenges in software development and programming language research for improving software quality and reliability. The problem in this area is to properly identify the defective source code with high accuracy. Developing a fault prediction model is a challenging problem, and many approaches have been proposed throughout history. The recent breakthrough in machine learning technologies, especially the development of deep learning techniques, has led to many problems being solved by these methods. Our survey focuses on the deep learning techniques for defect prediction. We analyze the recent works on the topic, study the methods for automatic learning of the semantic and structural features from the code, discuss the open problems and present the recent trends in the field.

Keywords: defect prediction; anomaly detection; program analysis; code understanding; neural networks; deep learning.

***How to cite this article**: Avinash Kori, Joy Bhattacharji, Prof. Anshul Jain, "Predicting Software Bugs using Deep Learning: A Comprehensive Review" Published in International Journal of Scientific Modern Research and Technology (IJSMRT), ISSN: 2582-8150, Volume-15, Issue-2, Number-1, May 2024, pp.1-9, URL: https://www.ijsmrt.com/wp-content/uploads/2024/06/ IJSMRT-24150201.pdf*

IJSMRT-24150201

I. INTRODUCTION

A software defect is, by definition, "an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be repaired or replaced," according to the IEEE Standard Classification for Software Anomalies [1]. Software bugs can result in a number of problems. Standard techniques for finding software defects include code reviews and manual testing. The significant drawback of these methods is their high time and energy requirements. Using automated techniques for Software Defect Prediction is one strategy to save expenses and enhance quality in software projects (SDP). For this reason, research on software bug prediction is essential in the fields of programming languages and software engineering. Finding the problematic code precisely (both in terms of recall and precision) is the aim. The advancement and improvement of machine learning has led to the automation of solutions for many difficulties. Advances in the fields of artificial neural networks and machine learning, along with the increasing power of modern computers (such supercomputers built on GPUs with AI acceleration modules), have given rise to new concepts like deep learning.

The fundamental idea is that complex issues can be solved by using a multi-layered artificial neural network, which can progressively extract higher-level properties from the original input. A possible answer to the problem of software defect prediction is provided by the development of representation-learning algorithms by researchers, which automatically learn semantic representations of programs and use this representation to identify the code that is prone to errors. It has been demonstrated that earlier methods that relied on explicit features—

like code metrics [2,3]—were not as effective as those that used these implicit aspects.

Software defect prediction is still in its infancy as a field, hence state-of-the-art surveys [3-5] only include a small portion of the most current research on cutting-edge techniques. Recent advances in natural language processing (NLP) and related fields have produced new, potent tools, such as the Transformer language models. These techniques were then successfully applied to software engineering projects. The goal of our survey is to present an overview of these advancements in light of the most current primary research that will be published in 2019–2021. This survey may be useful for scholars and experts in the fields of software defect prediction, code interpretation, and related fields.

Certain semantic errors may be difficult to find just by looking at the source code. For example, in [6], the bytecode of Kotlin programs that have been compiled is examined to identify compiler errors. In [7], the assembly code—which the compiler generates from the C source code—is utilized to reveal the behavior of the program and identify its flaws. Nevertheless, the main source of information for the fault prediction remains the source code. The main focus of this overview is on resources and techniques for researching program sources. Generally, when developing a defect prediction model, the subsequent steps are employed:

Getting source code samples from software project repositories is the initial step in preparing the dataset (or finding an appropriate existing dataset). Second, take functionality out of the code. Use the train dataset to teach the model. To evaluate the model's effectiveness, run tests on the test dataset.

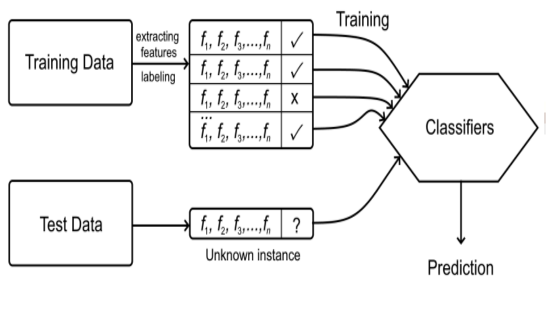

Figure 1: Scheme of the process of constructing the defect prediction model.

This is how the survey is laid out: In Section 2, we briefly outline our survey's methodology. The different deep learning approaches used for defect prediction are summarized in Section 3. The key challenges of the issue are outlined in Section 4. The most up-to-date research on defect prediction algorithms is presented in Section 5. In the last section, we discuss our predictions for the field's future.

2. RESEARCH QUESTIONS

Let's recap our survey's findings by developing some study questions:

- RQ1: Which deep learning methods have been successfully used for predicting software defects?
- RQ2: To what extent do these important aspects contribute to the problem's complexity?
- RQ3: What tendencies have been seen in the foundational work using deep learning for predicting software defects?
- RQ4: What Approaches Have Been Taken So Far?

We need a representation of the source code in order to modify it. Since most machine learning algorithms employ vectors, this representation should, on the one hand, be as straightforward as possible. However, the portrayal must include all relevant details. An "embedding" is a numerical vector that represents the code itself.

The code's source may be represented in a number of different ways. Furthermore, different granularities are required for various purposes, such as token-level embedding for code completion and function-level embedding for function clone detection.

Subsystem, component, file/class, method, and change embeddings are all employed for the software defect prediction issue (for more information on code embeddings, see [8,9]). The vector may be made using the constructed features as a starting point. In this method, the best characteristics are hand-picked by an expert (see, for example, [10,11]). The statistical properties of code, such its size, complexity, churn, and process metrics, are typical examples. The numerical vector may also be generated by interpreting the code itself. The code may be represented in a number of different ways.

Code tokens or characters are the norm [12]. Predicting what will come next is a common goal in training sequence-based neural networks. Abstract syntax trees (AST) [13] are another method for representing source code. Statements and operators are represented as tree nodes, with operands and values as leaves. Tree-based models are taught to make code predictions by creating new nodes in the tree while taking into consideration the current one. Using a classification technique to split the code into two groups—defect code and good code—is the standard method for defect prediction (see, for example, [14]).

However, methods relying on these hand-crafted characteristics often fail to accurately reflect the source code's syntax and semantics. If two pieces of code have the same structure and complexity but implement distinct functions, it might be difficult for standard code metrics to tell them apart. Traditional properties, such as the number of lines of code, the number of function calls, and the number of tokens, would stay the same, even if we switched several lines in the code fragments (see [2]). Therefore, the semantic information is more crucial than these measurements for fault prediction.

Instead than relying on explicitly hand-crafted features, modern methods often rely on extraction of the source code's implicit structure, syntactic, and semantic features. Deep Belief Networks (DBN), Convolutional Neural Networks (CNN), Long Short Term Memory (LSTM), and Transformer architecture are some of the most widely used deep learning approaches for software fault prediction.

3. DEEP BELIEF NETWORKS

Multi-layered neural networks provide the basis of Deep Belief Network generative models [15]. There is a single input layer, a single output layer, and numerous hidden layers in this network. A feature vector, representing the input layer's data, is generated by the output layer. The random links make up each successive layer. As can be seen in Figure 2, the DBN's defining characteristic is that nodes only communicate with those in the layers above and below them.

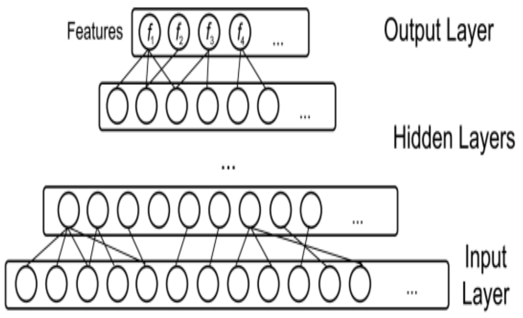

Figure 2: Architecture of the Deep Belief Network.

It's possible that [16] is one of the first attempts to combine AST with deep learning. The authors offer the method for predicting software defects at the level of code modifications. New expressive features are generated by the DBN (which is fed by the conventional code metrics) and used in the standard machine learning classifiers. Relationships are derived from conventional code metrics such the changed number of modules, folders, files, lines of code, and developer experience level. Following their success with the decision tree method, the authors presented an ensemble learning technique called "TLEL" [17].

Wang et al.'s [2,18] research also make use of the DBN, but in a somewhat different context. The authors have created a DBN to automatically learn a semantic characteristics from the source code, which can then be used for defect prediction based on code semantics. For both file-level and change-level prediction, the AST and source code modifications of the programs are utilized as input to the network. After extracting characteristics from source code files, the authors apply standard machine learning classifiers to determine whether or not the files include bugs. The fundamental problem with the DBN is that it does not accurately reflect the sequence in which statements are executed and functions are called.

4. LONG SHORT-TERM MEMORY

One sort of recurrent neural network designed specifically for handling data sequences is the Long Short Term Memory [19]. As can be seen in Figure 3, the LSTM network is built using LSTM nodes.

The unit's fundamental component is a memory cell, which stores values for both short and extended periods of time. This enables the LSTM-based models to extract the source code's long-range context information.

In work [11], an LSTM-based model was utilized to understand the syntactic and semantic components of source code. The suggested method converts code into a feature vector and a token state encoding the token's semantic information by feeding it into a Long Short-Term Memory (LSTM) system. A further model, the Tree-LSTM, was built using the AST format as its input [20].

In [21], a neural bug-finding approach is presented. The authors use a binary classifier, trained using samples of both flawed and perfect code, to identify errors. The authors use preexisting static bug detection tools to classify issues into appropriate categories in order to generate a labeled dataset. Using the one-hot encoding for each token, the code is transformed from its tokens sequence representation into a real-value vector. Then, an LSTM-based network acting as a model in both directions is used.

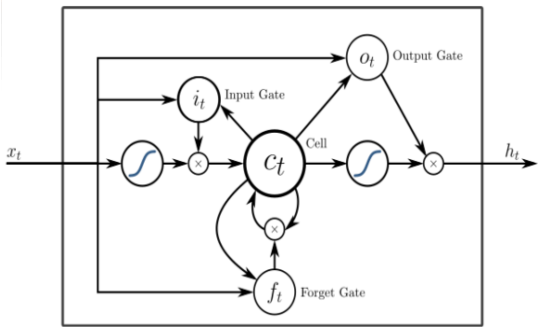

Figure 3: Scheme of the LSTM unit

The authors of [22] offer a model for defect prediction based on the representation of AST path pairs. The AST route is retrieved as a mixture of symbol and control sequence so that the code may be processed. A route vector is generated by feeding these sequences into a Bi-LSTM network. The global attention method is then used to aggregate all the vectors into a single one that represents the complete code snippet. The completed embedding representations are then used as a classification tool.

5. CONVOLUTIONAL NEURAL NETWORKS

When it comes to processing data with a mesh-like structure, the Convolutional Neural Networks [23] are the way to go. There are two distinguishing characteristics of this network. First, the network as a whole follows the same structure as the local nodes' connections. The network is able to learn the structural context of the code in the near term. Second, all the settings are the same across the board. The network may acquire knowledge about the code element regardless of where it occurs in the code. Figure 4 depicts the overall CNN design.

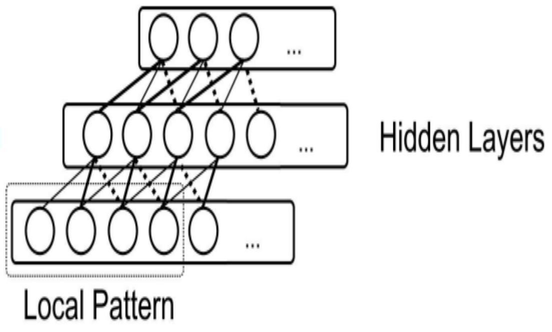

Figure 4: Architecture of the Convolutional Neural Network.

The model using the CNN architecture is shown in reference [24]. Token vectors are parsed from the AST and then transformed to numbers. Next, a CNN is trained using these vectors as input. The retrieved semantic and structural variables and code metrics are then employed in a logistic regression for software fault prediction.

Commit messages and code modifications are used as inputs into a deep learning model that predicts errors [25]. In this case, the CNN serves as the basis for the model. Convolutional network layers are used to analyze the code modifications and commit message, and a feature combination network layer combines the resulting embedding vectors. We see a proposal for yet another deep learning-based model for fault prediction. Triplet loss and weighted cross-entropy loss are used in the training of the neural network. As a kind of classifier, the random forest is used.

Machine learning methods that rely on neural networks find it difficult to tackle the challenge of software fault prediction.

A. Lack of Data

The absence of big, publicly accessible labelled datasets specifically for fault prediction is a significant obstacle. The issue may be solved by using the pre-trained contextual embeddings. Using the self-supervised goals such masked language modelling, next sentence prediction, and substituted token recognition, the language model is pre-trained on a large unlabeled source code corpus.

Table 1: presents the popular unlabeled code datasets suitable for this task.

Dataset	Content	Size	Used in Tasks
Bigquery github repos [3]	Python source code	4 M files	Pre-training CuBERT model
Py150 [4]	Python source code, AST	8423 repos, 149,993 files	Fine-tuning CuBERT model
Js150 [5]	Javascript source code, AST	150,000 source files	Code Summarization; Defect Prediction
Datasets for [6]	Java source code	9500 projects, 16 M samples in the largest one	Code summarization
GitHub Java Corpus [8]	Java source code	11,000 projects	Language Modelling
CodeNN Dataset [9]	C# source code and summaries	66,015 fragments	Code Captioning
Dataset for [6]	Kotlin source code, AST, bytecode	47,751 repos, 932,548 files, 4,044,790 functions	Anomaly detection, defect prediction
Dataset for [10]	C# source code	29 projects, 2.9 M lines of code	Variable Misuse detection

Smaller labelled datasets may be used to fine-tune the pre-trained model for defect prediction. Table 2 provides a catalogue of open-source datasets for defect prediction. These datasets often consist of pairs of good and bad snippets of code.

Table 2: List of labelled datasets.

Dataset	Content	Size	Used in Tasks
SEIP Lab Software Defect Prediction Data [11]	Complexity metrics	5 subsequent releases of 3 projects from the Java Eclipse community	Data collection and linking
PROMISE Software Enginee	Numeric metrics; reported defects	15,000 modules	Defect prediction

ring Repository [12]	(false/true)		
NASA Defect Dataset [13]	Numeric metrics; reported defects (false/true)	51,000 modules	Defect prediction
REPD datasets [15]	Numeric metrics, semantic features, reported defects	10,885 fragments in the largest one	Defect prediction
GPHR [16]	Java code and metrics	3526 pairs of fragments, buggy and fixed, code metrics	Defect prediction
BugHunter [17]	Java source code; metrics; fix-inducing commit; number of reported bugs	159 k pairs for 3 granularity levels (file/class/method), 15 projects	Analyzing the importance of complexity metrics
GitHub Bug DataSet [18]	Java source code; code metrics; number of reported bugs and vulnerabilities	15 projects; 183 k classes	Bug prediction
Unified Bug Dataset [19]	Java source code; code metrics; number of reported bugs	47,618 classes; 43,744 files	Bug prediction
Neural Code Translator Dataset [20]	Pairs of buggy and fixed abstracted method-level fragments 46 k pairs of small fragments	(under 50 tokens), 50 k pairs of medium fragments (under 100 tokens)	Code refinement

Real-world code projects sometimes have an uneven distribution of classes, which, together with other considerations, increases the challenge of creating datasets. There are often fewer incorrect files or methods than right ones in a project. The result may be that the standard classifiers would identify the larger group (the proper code) but fail to recognize the much smaller group (the defect-prone code). The model's performance will suffer as a result of this.

Several oversampling strategies are presented as a means of correcting this discrepancy. The authors developed combined methods. It uses Synthetic Minority Over-Sampling Methods (SMOTE and SMOTUNED) to prepare datasets and ensemble methods to sort out bugs and good programs. The fraction of accurate and faulty code in each project in the dataset is considered by the authors of [22]. The components of the smaller class are duplicated in order to achieve class balance.

B. Lack of Context

The context in which the code exists is also problematic. In contrast to natural texts, a code element's dependencies may extend to neighboring code fragments or even farther afield. It is also frequently difficult to determine whether the code piece is flawed without knowing the larger context in which it is used. It may be challenging to distil the core of a defect from a dataset that consists of pairs of bugged and corrected code snippets.

The Transformer network-based methods were developed for natural language processing issues characterized by a high degree of locality of reference in the underlying data. A token's immediate surroundings reveal a great deal about it. As a result, such models often interpret the code's source as a list of tokens.

5. CONCLUSIONS

Predicting faulty code is a key obstacle in the field of software engineering today. Recently developed multi-layered neural networks and deep learning algorithms offer potent ways for representing source code in a way that captures its semantic and structural information using learning algorithms.

Using deep learning approaches, such as the Transformer architectures, this survey details the most recent findings in software fault prediction research. We identify the primary challenges of the defect prediction problem as an absence of data and a high degree of contextual complexity, and we propose potential solutions to these issues.

We think that the following concepts, which take into consideration the most recent developments in machine learning approaches for the software defect prediction issue, will contribute significantly to the advancement of this discipline.

• Self-supervised training on vast corpora of unlabeled data may be used to minimize the size of the required labelled datasets. In addition, the unlabeled data must be used for the pre-training of associated tasks, which contributes to the trained models' enhanced depth and breadth of knowledge of the original code. As a result, the underlying flaws may be identified.

• We are currently seeing the effective translation of these approaches to tackle diverse code comprehension challenges, capitalizing on the most recent breakthroughs in machine learning techniques in natural language processing in programming languages. To better consider the code context while searching for faults, one may optimize the transformers' self-attention mechanism so that they can be used across extended sequences.

• It's fairly uncommon for bugs to affect many related functions or lines of code, and to have multiple potential solutions. It's possible, for instance, to correct a defect either inside the function itself or at the point when it's called. As a result, the error can no longer be located at a particular line number in the source code. Even if a bug isn't initially present in a single line of code, it may become a defect later on. The original implementation may no longer be suitable due to the fact that the code's original intent has changed as a result of new circumstances.

The idea of a flaw becomes muddled as a result of all this. The terms "potentially defective" and "strange" code are so introduced. Finding an unusual code and honing existing ones are two examples of interesting challenges in this area. Good representations of the code and updates to the code, taking into consideration the structure and context of the source code, are necessary for these tasks.

Identifying a best-performing state-of-the-art model is challenging. There are no agreed-upon, industry-wide measures by which this issue is measured, and academics are using a wide variety of measurements and datasets in their investigations. As a consequence, comparing the experimental outcomes of the foundational papers is problematic. While state-of-the-art deep learning algorithms often outperform baseline deep learning and classic metrics-based ones (achieving the rise of F1 from 60% to 80% in certain circumstances), extant comparative studies such as suggest otherwise. No method provides a level of reliability in recall, precision, and accuracy that is adequate for real-world use. Defect prediction is therefore still an unsolved issue.

REFERENCES

[1] Jitimon Angskun, Suda Tipprasert and Thara Angskun, "Big data analytics on social networks for real-time depression detection", Journal of Big Data, 2022.

[2] Md. Rafidul Hasan Khan; Umme Sunzida Afroz; Abu Kaisar Mohammad Masum; Sheikh Abujar; Syed Akhter Hossain, "Sentiment Analysis from Bengali Depression Dataset using Machine Learning", 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 2020.

[3] Tanna D, Dudhane M, Sardar A. Deshpande K, Deshmukh N., "Sentiment analysis on social media for emotion classification", In: International Conference on intelligent computing and control systems, 2020.

[4] Md Kamrul Hasan, Hasan Mahmud, Ahmed Al Marouf, "Comparative Analysis of Feature Selection Algorithms for Computational Personality Prediction From Social Media", IEEE Transactions on Computational Social Systems, 2020.

[5] Yang X, McEwen R, Ong LR, Zihayat M., "A big data analytics framework for detecting user-level depression from social networks", Int J Inf Manag. 2020.

[6] Lyua YW, Chow JC-C, Hwang J-J, "Exploring public attitudes of child abuse in mainland China: a sentiment analysis of China's social media", Weibo. Child Youth Serv Rev. 2020.

[7] Chen B, Cheng L, Chen R, Huang Q, Phoebe Chen Y-P. Deep neural networks for multiclass sentiment classification. In: IEEE 20th International Conference on high performance computing and communications, IEEE 16th International Conference on Smart City, IEEE 4th International Conference on Data Science and Systems 2018; pp. 854–59.

[8] Sethi M, Pande S, Trar P, Soni P. Sentiment identification in COVID-19 specific tweets. In: International Conference on electronics and sustainable communication systems (ICESC 2020), pp. 509–16, https://doi.org/10.1109/ICESC48915.2020.915567 4.

[9] Kundale JU, Kulkarni NJ. Language independent multi-class sentiment analysis. In: 5th International Conference on computing communication control and automation (ICCUBEA), 2019; pp. 1–7, https://doi.org/10.1109/ICCUBEA47591.2019.912 8383.

[10] Ruz GA, Henriquez PA, Mascareno A. Sentiment analysis of Twitter data during critical events through Bayesian networks classifers. Future Gener Comput Syst. 2020;106:92–104.

[11] Yang X, McEwen R, Ong LR, Zihayat M. A big data analytics framework for detecting user-level depression from social networks. Int J Inf Manag. 2020;54:102141.

[12] Tao X, Dharmalingam R, Zhang J, Zhou X, Li L, Gururajan R. Twitter analysis for depression on social networks based on sentiment and stress. In: 6th International Conference on behavioral, economic and socio-cultural computing, 2019; pp. 1-4, https:// doi.org/10.1109/BESC48373.2019.8963550.

[13] Tanna D, Dudhane M, Sardar A. Deshpande K, Deshmukh N. Sentiment analysis on social media for emotion classifcation. In: International Conference on intelligent computing and control systems (ICICCS 2020), pp. 911–15, https://doi.org/10.1109/ ICICCS48265.2020.9121057.

[14] Arora P, Arora P. Mining Twitter data for depression detection. In: IEEE International Conference on signal processing and communication (ICSC), 2019; pp. 186–89, https://doi.org/10.1109/ ICSC45622.2019.8938353.

[15] Chen Y, Zhou B, Zhang W, Gong W, Sun G. Sentiment analysis based on deep learning and its application in screening for perinatal depression. In: IEEE Third International Conference on data science in cyberspace. 2018; pp. 451–6. https://doi.org/ 10.1109/DSC.2018.00073.

[16] Uddin AH, Bapery D, Arif ASM. Depression analysis from social media data in Bangla language using long short term memory (LSTM) recurrent neural network technique. In: International

Conference on computer, communication, chemical, materials and electronic engineering (IC4ME2), 11–12 July, 2019; pp. 1-4, https://doi.org/10.1109/IC4ME247184.2019.9036528.

[17] Cheng L-C, Tsai S-L. Deep learning for automated sentiment analysis of social media. In: IEEE/ACM International Conference on advances in social networks analysis and mining. 2019; pp. 1001–4. https://doi.org/10.1145/3341161.3344821.

[18] Al Asad N, Pranto MAM, Afreen S, Islam MM. Depression detection by analyzing social media posts of user. In: IEEE International Conference on signal processing, information, communication & systems(SPICSCON). Dhaka, Bangladesh, 2019; pp. 13–17, https://doi.org/10.1109/SPICSCON48833.2019.9065101.

[19] Lyua YW, Chow JC-C, Hwang J-J. Exploring public attitudes of child abuse in mainland China: a sentiment analysis of China's social media Weibo. Child Youth Serv Rev. 2020;116:102520.

[20] Abid F, Li C, Alam M. Multi-source social media data sentiment analysis using bidirectional recurrent convolutional neural networks. Comput Commun. 2020;157:102–15.

[21] Hammou BA, Lahcen AA, Mouline S. Towards a real-time processing framework based on improved distributed recurrent neural network variants with fastText for social big data analytics. Inf Process Manag. 2020;57:102122.

[22] Tadessi MM, Lin H, Xu B, Yang L. Detection of depressionrelated posts in reddit social media forum. IEEE Access. 2019;7:44883–93. https://doi.org/10.1109/ACCESS.2019. 2909180.

[23] Trotzek M, Koitka S, Friedrich CM. Utilizing neural networks and linguistic metadata for early detection of depression indications in text sequences. IEEE Trans Knowl Data Eng. 2018;32:588–601.

[24] Tariq S, Akhtar N, Afzal H, Khalid S, Mufti MR, Hussain S, Habib A, Ahmad G. A novel co-training based approach for the classifcation of mental illnesses using Social media posts. IEEE Access. 2019;7:166165–72. https://doi.org/10.1109/ACCESS.2019.2953087.

[25] Rao G, Zhang Y, Zhang L, Cong Q, Feng Z. MGL-CNN: a hierarchical posts representations model for identifying depressed individuals in online forums. IEEE Access. 2020;8:32395–403. https://doi.org/10.1109/ACCESS.2020.297373.