How to Mitigate the Incident? An Effective Troubleshooting Guide Recommendation Technique for Online Service Systems

Jiajun Jiang*
College of Intelligence and Computing, Tianjin University
Tianjin 300350, China
jiangjiajun@tju.edu.cn

Weihai Lu
School of Software and Microelectronics, Peking University
Beijing 100871, China
luweihai@pku.edu.cn

Junjie Chen
College of Intelligence and Computing, Tianjin University
Tianjin 300350, China
junjiechen@tju.edu.cn

Qingwei Lin
Microsoft Research
Beijing 100080, China
qlin@microsoft.com

Pu Zhao
Microsoft Research
Beijing 100080, China
pu.zhao@microsoft.com

Yu Kang
Microsoft Research
Beijing 100080, China
kay@microsoft.com

Hongyu Zhang
The University of Newcastle
NSW 2308, Australia
hongyu.zhang@newcastle.edu.au

Yingfei Xiong
Key Laboratory of High Confidence Software Technologies(MoE), DCST
PKU, Beijing 100871, China
xiongyf@pku.edu.cn

Feng Gao
Microsoft Azure
Redmond, WA 98052, USA
fgao@microsoft.com

Zhangwei Xu
Microsoft Azure
Redmond, WA 98052, USA
zhangxu@microsoft.com

Yingnong Dang
Microsoft Azure
Redmond, WA 98052, USA
yidang@microsoft.com

Dongmei Zhang
Microsoft Research
Beijing 100080, China
dongmeiz@microsoft.com

ABSTRACT

In recent years, more and more traditional shrink-wrapped software is provided as 7x24 online services. Incidents (events that lead to service disruptions or outages) could affect service availability and cause great financial loss. Therefore, mitigating the incidents is important and time critical. In practice, a document describing a mitigation process, called a *troubleshooting guide (TSG)*, is usually used to reduce the Time To Mitigate (TTM). To investigate the usage of TSGs in real-world online services, we conduct the first empirical study on 18 real-world, large-scale online service systems in Microsoft. We analyze the distribution and characteristics of TSGs among all incident records in the past two years. According to our study, 27.2% incidents have TSG records and 36.2% of them occurred at least twice. Besides, on average developers spend around 36.3% of the entire mitigation time on locating the desired TSGs. Our study shows that incidents could occur repeatedly and TSGs could be reused to facilitate incident mitigation. Motivated by our empirical study, we propose an automated TSG recommendation approach, DeepRmd, by leveraging the textual similarity between incident description and its corresponding TSG using deep learning techniques. We evaluate the effectiveness of DeepRmd on 18 online service systems. The results show that DeepRmd can recommend the correct TSG as the Top 1 returned result for 80.3% incidents, which significantly outperforms two baseline approaches.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Search-based software engineering**.

KEYWORDS

Incident management, incident mitigation, troubleshooting guide, online service systems

ACM Reference Format:
Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2020. How to Mitigate the Incident? An Effective Troubleshooting Guide Recommendation Technique for Online Service Systems. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3368089.3417054

1 INTRODUCTION

Recently, with the growth of requirements for complex software functionality, the volume of software increases rapidly. As a consequence, traditional shrink-wrapped software is moving to online services, which have become increasingly popular especially in

*This work was done at Microsoft Research (Beijing, China). Jiajun Jiang and Weihai Lu contributed equally to this research. Qingwei Lin is the corresponding author.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417054

industry, such as Microsoft Office 365, Google Docs, etc. In order to ensure the service quality of online systems, various control measurements have been employed to guarantee that services are running normally [8, 10, 34, 35, 50, 51, 53, 54]. However, incidents [34] are still common in practice and can potentially cause massive financial loss [4, 23]. For example, a study conducted by Ponemon Institute in the U.S. reported that the average cost of a data center outage has steadily increased up to 38% from 2011 to 2016 [1], and the average cost is up to $740,357 in 2016. Additionally, as reported that due to the service downtime on Amazon's Annual Prime Day (its biggest sale event) in 2018, the estimated cost is up to $100 million per hour [2].

Therefore, when an incident of online services occurs, it is vital to mitigate it as soon as possible. Typically, incidents are reported automatically by systems. In practice, an incident management system continuously detects incidents of services via a set of monitors. When an incident is detected, a set of On-Call Engineers (OCEs) will be informed automatically, where the investigation of the incident starts. As the downtime of services is vital, the investigation process should be as quick as possible to minimize its impact. In this process, OCEs need to understand the reasons of the incident or identify the root cause, and then resolve it to recover normal services. However, the root causes of incidents are diverse, such as UPS system failures, human errors and so forth, and thus it usually costs unacceptable time to resolve it online. Therefore, OCEs need to mitigate the incident and bring the service back to normal first, and then resolve the root cause offline later.

In practice, since the responsible developers are frequently changing and the on-call engineers may not know how to mitigate the incident immediately. As a result, the mitigation process is also costly. In fact, similar incidents may occur repetitively due to inevitable reasons (e.g., power loss), where the mitigation process should be similar or even the same (e.g., restarting services). Therefore, a good practice should record the mitigation process when the incidents first occur (referred to as a *troubleshooting guide* or *TSG*), and later reuse the TSG when similar incidents occur. Ideally, when a new incident comes, OCEs can search the corresponding TSG according to its symptoms to mitigate the incident as soon as possible. However, since there are too many TSGs and the symptoms may vary, it is not easy to find the desired one and potentially may cause longer mitigation time.

To better understand the usage of TSGs and their characteristics in industrial practice, we conducted the first large-scale empirical study on industrial systems from Microsoft. According to the analysis, a significant portion (i.e., 27.2%) of incidents have corresponding TSGs, and some of them repetitively occurred. For example, around 36.2% TSGs occurred at least twice in the past two years. Besides, developers took non-negligible time, i.e., on average 36.3% of the entire mitigation time, to find the desired TSG for an incident. Additionally, by further analyzing the characteristics of incidents and corresponding TSGs, we find that the textual similarity between the description of the incident and its TSG holds to some extent.

This observation of TSG repetitiveness from our empirical study reveals that designing an effective TSG recommendation approach has great potential to speed up the investigation process of incidents and reduce financial loss in industry. As a consequence, according to the insights learned from the analyzing results, we propose a TSG recommendation approach, named DeepRmd, by leveraging deep learning techniques. Since there is no previous study on this topic, we employ a traditional TF-IDF approach [38], which is widely used for computing textual similarity in the literature [5, 39], and shift a deep-learning-based technique, DeepCS [21], from a similar application domain (i.e., code search) to our scenario as two baseline approaches. We evaluate and compare the effectiveness of our approach with the baseline techniques on the real-world online service systems. The results demonstrate that our approach is able to achieve the recommendation accuracy of 80.3%, significantly outperforming the baseline techniques with up to 104.8% and 47.6% increases, respectively. Besides, we further investigate the contribution of each component of our approach via a set of comparison experiments. The results demonstrate the effectiveness of each individual component. Additionally, since the length of TSG text varies, we also investigate its impact on the final result by changing the length of TSG text used for learning and prediction.

In summary, the major contributions of this paper are as follows:

- The first large-scale empirical study on the usage and characteristics of TSGs for online service systems in industrial practice.
- A novel TSG recommendation approach (named DeepRmd) by leveraging the textual similarity between incident description and its corresponding TSG using deep learning techniques, which is effective and significantly outperforms the baseline techniques.

The remainder of the paper is organized as follows. Section 2 introduces the empirical study and demonstrates the analysis results. Section 3 introduces our new approach of TSG recommendation in detail and Section 4 presents the evaluation results. Section 5 introduces the lessons learned from our empirical analysis and evaluation. Section 6 and Section 7 present the threats to validity and related work, respectively. Finally, we conclude the paper in Section 8.

2 EMPIRICAL STUDY

To investigate the importance of recommending TSGs to developers, we perform an empirical study on the characteristics of TSGs and their usage in online service systems. Though it is intuitive that TSG recommendation should be helpful, the empirical study on real-world online systems can provide statistical supports to the intuition and further motivate our TSG recommendation approach.

2.1 Data Collection

To perform our empirical study, we statistically analyze 18 large-scale industrial online service systems from Microsoft, some of which are widely used products around the world, such as Microsoft Azure, Visual Studio, etc. We collect all incident records in the past two years of those systems. For each incident, the major content includes a title (or description), the corresponding TSG (empty if it does not have), related service and environment, as well as the timestamps of incident creation, assigning the corresponding TSG and finishing mitigation. The title is a short summary (i.e., one or two sentences) of an incident that may include impacted services and related resources. The TSG is usually an online document which records the detailed description of a mitigation process.

A typical TSG document contains dozens to thousands of lines of texts. The related service field records the service system on which the incident was introduced. The environment field denotes the phase of services where the incident was reported (e.g., the testing environment indicates that the incident occurred during the development of the system, while the production environment indicates that the system was already online). Finally, the timestamp of incident creation indicates the time when it was reported. The timestamp of assigning TSG denotes the time when the associated TSG was assigned to the incident while the timestamp of finishing mitigation is the time when the developers tagged the incident as successfully mitigated. In our analysis, we discard those records that under testing environment to alleviate the impact of irrelevant data. As a result, we collect about 20GB data (due to the confidential policy of Microsoft, we hide the details of the incidents, e.g., incident categories) and 1500 TSGs.

Incident description:
[AX] - Watch Dog RuleName DatabaseSpaceUsedRule 10PercentRemaining for Tenant 9a083aab-e8d6-459d-8407-xxxxx ...

Corresponding TSG:
Watchdog Rule Failure: Database Space Used
Symptoms:
Run the following query to get the current usage and free space details about the database.
*If the used percentage (USEDMB/TOTALMB*100) is greater than xx – it's bad, do the following*
...
SELECT [t].[name] AS [Table], [i].[name] AS [Index], [p].[partition_number] AS [Partition], [p].[data_compression_desc] AS [Compression] FROM [sys]
...

Figure 1: An example of incident description and its corresponding TSG (removing sensitive information).

Figure 1 shows the excerpt of an incident description, which was reported at 23:43 (incident creation). Its corresponding TSG shown in the figure was attached to the incident at 23:49 (assigning TSG). Finally, the incident was mitigated at 04:33 on the next day (finishing mitigation). From the descriptions, we can see that the incident is associated with the usage of a certain database and encountered by a tenant with a certain id. The corresponding TSG first describes the symptoms and then the mitigation process in detail, such as query a database to check the status of systems. Typically, the symptom description of the TSG is similar to that of a corresponding incident, which is also the intuition of our recommendation approach.

2.2 Empirical Analysis

In this section, we report our empirical results in detail. Especially, we mainly focus on the following research questions.

Q1: What is the *ratio* of TSG usage in online service systems?
Q2: What is the impact of TSG on *incident mitigation time*?
Q3: What are the *characteristics* of TSG usage in practice?

2.2.1 Frequency of TSGs. We study the frequency of using TSGs in online service systems. The results are presented in Figure 2. In the figure, *x-axis* denotes the names of systems while *y-axis* denotes the percentage of incidents that have TSG records in the history (past two years). Due to the confidentiality policy of Microsoft, we hide the details of those services and use "Si" to represent the i^{th} service. On average, 27.2% incidents have recorded the corresponding TSGs, indicating that TSGs are useful in practice to some extent. Please note that this ratio is an approximation since we cannot guarantee that all developers have recorded the TSG information in the mitigation process even if they have actually used TSGs and thus this ratio should be the lower bound. On the other hand, since currently there is no effective TSG recommendation technique in the mitigation process, the frequency of TSG usage is affected. We will further discuss this issue in Section 6.

Furthermore, from the figure we can find that the ratio of incidents varies greatly among different services. For example, for service S2, a large portion (i.e., 85.2%) of incidents have TSG records, while only a small portion (i.e., 1.3%) for service S17. As a result, we further investigate the reasons for the divergence. First, those services with smaller portions of TSGs are usually background services (such as S6, S13, etc.), which have small or even no impact on user experience. Therefore, these incidents are not very time urgent and thus can be mitigated by experienced developers later without TSGs. Second, a large portion of incidents may occur repetitively but last for a very short time (called *transient* incidents). Such kind of incidents usually do not require human efforts to mitigate as they have small impact on users and will disappear soon, such as incidents in S10, and S17.

> **Observation 1.** *On average, developers use TSGs for around 27.2% incidents (lower bound) of online service systems. The results indicate that some teams frequently use TSGs while some do not. It is desirable to facilitate the use of TSGs by automatically recommending TSGs.*

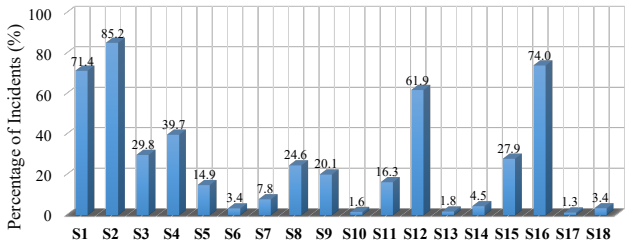

Figure 2: The percentage of incidents that have TSGs for each service.

2.2.2 Impact on Mitigation Time. To understand whether TSGs impact the efficiency of incident mitigation, we compare the mitigation time (i.e., Time to Mitigate or TTM) for incidents that have and do not have TSG records. Particularly, according to Figure 2, the proportions of incidents that have TSGs are too small for some services, which may not be representative and tend to be biased

without statistical significance. To alleviate this effects, we only focus on those services from which at least 10% of the incidents have associated TSG records. Finally, it leaves us 11 services for analysis and the results are reported in Figure 3, where the *x-axis* represents the service while the *y-axis* denotes the multiple of average TTM for incidents without TSG record against that for incidents having associated TSGs. On average, compared with those using TSGs, developers spent 6 times as much time to mitigate an incident as there is no TSG used. Additionally, from the figure we can find that TSGs can help shorten TTM for 10 out of 11 services. That indicates it is useful in practice to speed up the mitigation process of incidents. For example, the time saved to mitigate an incident can be as high as around 20x, which is significant to reduce TTM and thus financial loss incurred by the incident. However, there is an exception, where the mitigation time for S2 became longer when using TSGs. We further analyze these cases and find that among all the incidents which do not have TSGs, around 87% of them (via manually inspecting 100 randomly sampled cases) have very small impact on users and will automatically come to normal service within a short time. As a result, the monitoring system will report the normal state in time, which looks like "*Incident is mitigated because the watchdogs associated with this incident have reported healthy at least 5 times*". Therefore, such kinds of incidents do not cause too much time for mitigation. On the other hand, incidents with TSGs need to be tackled by developers, where usually more time will be spent.

> **Observation 2.** *TSGs can achieve around 5x speedup to incident mitigation, indicating their effectiveness to reduce TTM in practice.*

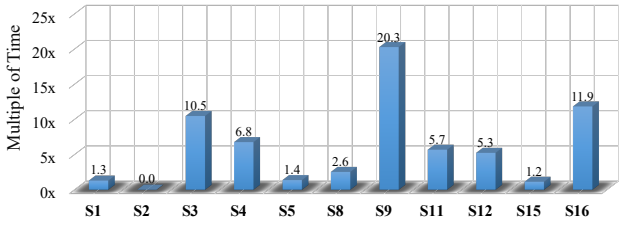

Figure 3: The multiple of average mitigation time of incidents without TSGs compared with that of incidents with associated TSG records.

2.2.3 Characteristics of TSG Usage. According to the results shown in previous sections, developers tend to use TSGs for a non-negligible portion of incidents, which can help reduce the mitigation time of incidents. Since TSGs are useful and helpful to developers, we naturally raise a question – is it necessary and practical to recommend TSGs for developers? To answer this question, we perform a further analysis from the following two perspectives.

- How long does it take to locate the desired TSG?
- Are TSGs used repetitively?

These two aspects are important as they correspond to the necessity and practicality of TSG recommendation. For example, if it was easy (i.e., spending a little time) to locate the desired TSG for developers, the gain of TSG recommendation would be minimal since the saved time could be ignored. On the other hand, if developers spent much time on locating the corresponding TSGs, it would be very helpful to speed up the incident mitigation process. Likewise, the recommendation depends on the repetitive occurrence of TSGs for practical use. Therefore, we first conduct a statistical analysis of the time to locate the corresponding TSGs. However, since it is impossible to obtain the accurate time of TSG localization and incident mitigation due to human factors (e.g., non-continuous working on the incident). As a result, we use the time when they were recorded to the incident management system as an approximation. We will discuss this further in Section 6. In detail, we use t_1, t_2 and t_3 to respectively represent the recorded timestamps of incident creation, assigning the corresponding TSG and finishing mitigation process (*ref.* Section 2.1). Then, the TSG localization and mitigation time can be computed by $t_2 - t_1$ and $t_3 - t_1$, respectively. Figure 4 presents the analysis results. In the figure, the *y-axis* indicates the average percentage of time to locate the desired TSG among the entire mitigation time, i.e., $\frac{t_2-t_1}{t_3-t_1}$.

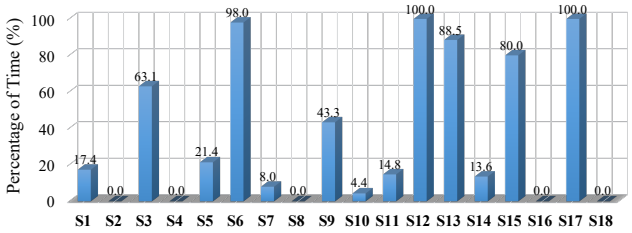

Figure 4: The ratio of time for finding the desired TSG to the complete mitigation time.

On average, 36.3% of the mitigation time is spent on locating the TSGs, which is a significant portion of time in this process. Furthermore, from the figure, up to 80% or even a larger portion of time is used for TSG localization in five services, i.e., S6, S12, S13, S15 and S17, demonstrating the localization process is the major time-consuming part to mitigate the incidents. In other words, if the corresponding TSG is found, the mitigation process should be much easier and cost a little time. As a result, correctly recommending TSGs for developers is useful and necessary. However, some special cases should be noticed in the figure, where little time (i.e., close to 0) is used to locate the desirable TSGs, i.e., for services S2, S4, S8, S16 and S18. The reason is that those services have mature mitigation maintenance systems, where engineers clearly classify different kinds of incidents, and allocate different monitors to capture each kind of incidents and report the corresponding TSGs together with the incidents. However, it depends on a well-designed system which can correctly identify the reasons for reported incidents with particular monitors. As for the other services, incidents caused by different reasons may be reported by the same monitors, making it impossible to correctly recommend TSGs. However, on the one

hand, it is not easy to design such a system because developers need to not only know most of possible incidents but also seed enough monitors to capture them online. On the other hand, many services in Microsoft that have been deployed were not designed with such a mature monitoring system (see Figure 4) in the beginning and it is hard to re-implement them within a short time. As a result, most of the time developers need to find the desired TSGs manually.

> **Observation 3.** *Developers usually spend a significant portion of time (i.e., 36.3% on average) on locating the desired TSGs in the incident mitigation process.*

To investigate the repetitiveness of TSGs, we collect all incident TSGs and compute the number of times each TSG appeared. Especially, we rule out those TSGs that can be handled well by the mature mitigation maintenance systems since they can be recommended by the systems automatically and do not need external recommendations, e.g.,S2 and S4. The result shows that 36.2% of TSGs repetitively appeared at least twice in the historical records and some of them were even shared by hundreds of incidents. That is to say, for only one TSG, hundreds of incidents can be potentially sped up by an effective recommendation technique. However, for the other 63.8% TSGs, they were rarely shared as only used by one incident in the past two-year records. To investigate the reasons for those TSGs, we analyze their features and find that some of them are related to one-time-only disruptions, such as source code defects or configuration errors, which can be avoided once repaired or mitigated. Additionally, we further randomly sample ten of those TSGs which appeared only once and manually check whether there exist other related incidents. This is because developers may miss to record corresponding TSGs as it is not a must in Microsoft. The results demonstrate that some TSGs (2 out of ten) in fact should be repetitive as there exist similar or even the same (in terms of descriptions) incidents corresponding to the TSGs, but they were assigned no TSG in our dataset. In other words, at least 36.2% TSGs (lower bound) have the potential to be automatically recommended to speed up the incident mitigation process. Please note, for the other 63.8% TSGs, it is still possible that they are repetitively appearing in the future and can be recommended eventually. Furthermore, the TSG recommendation technique potentially can facilitate the usages of TSGs in practice by saving TTM.

> **Observation 4.** *A significant portion of TSGs (i.e., 36.2%) were repetitively used to mitigate incidents, indicating the potential of reducing the incident mitigation time via automated TSG recommendation.*

3 TSG RECOMMENDATION TECHNIQUE

According to the empirical study in Section 2, it is necessary to recommend TSGs for developers to speed up the incident mitigation process. In this section, we aim at this target and propose a TSG recommendation approach by leveraging deep learning techniques, named DeepRmd. Since the number of TSGs is not fixed, i.e., new TSGs can be added to the system continuously, traditional classification techniques do not conform to this application scenario.

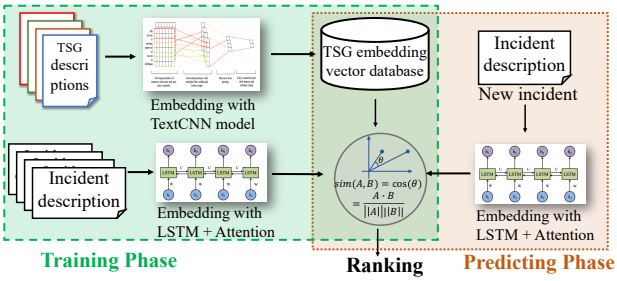

Figure 5: The architecture of DeepRmd

Therefore, via making use of the textual similarity between the incident description and its corresponding TSGs (e.g., the example shown in Figure 1), we consider it as an information retrieval problem, where the query is the incident report and the answer is the corresponding TSG.

3.1 Architecture

Inspired by existing joint embedding techniques [16, 17, 21, 28, 46], we propose a novel deep neural network for TSG recommendation, called DeepRmd. The architecture of our approach is shown in Figure 5, which consists of two phases, i.e., training phase and predicting phase. For each phase, we first compute the embeddings of TSG descriptions and incident descriptions, respectively, and make their embeddings (feature vectors) in the same length. Finally, we leverage the *cosine* function to compute the similarity between the embedded vectors for joint learning (training phase) and ranking (predicting phase).

Specifically, we first utilize a widely used pre-trained model [14] to perform word segmentation for both incident and TSG descriptions. Then, we employ a bidirectional LSTM model [19, 20, 42] to compute the embeddings of incident descriptions and a TextCNN model [30, 49] to compute the embeddings of TSG descriptions. Please note that the embeddings of TSG descriptions can be reused for future prediction. That is, when a new incident comes, we compute its embedding first and then calculate the similarity with each of the TSG embeddings via the *cosine* function. Particularly, to speed up the computation, we adopt the random projection technique [13] implemented by Bernhardsson et al. [3] for dimension reduction. Eventually, all candidate TSGs are ranked based on the similarities. Next, we will introduce these two kinds of embeddings in detail.

3.2 Embedding of Incident

To compute the embeddings of incidents, we employ a Recurrent Neural Network (RNN), which is a popular architecture of Neural Network and is widely used in the field of Natural Language Processing (NLP) [37]. The main difference of RNN with other neural networks is that RNN is good at handling sequential or contextual data by viewing them as different time-evolving states.

However, standard RNN performs just fine when dealing with short-term dependencies, that is, when the sentence is short. However, it has a primary disadvantage that the relevant information is separated if the sentence consists of a large number of words.

In this case, the performance of the network may downgrade due to vanishing gradient [7]. To solve this problem, a variant of RNN called Long Short Term Memory (LSTM) is proposed to preserve the long-term dependency information [22]. Therefore, in our approach, we leverage a bidirectional LSTM [19, 20, 42] model for the embeddings of incident descriptions.

Despite the LSTM model can alleviate the dependency problem of standard RNN model, it still views all words equally and hardly identifies important keywords in a sentence [6]. As a result, in our approach, we further leverage the attention mechanism to improve the effectiveness of LSTM model. More specifically, for each word in the textual description of an incident, we compute the word vector with the LSTM model, and in the meantime we employ the attention cell to highlight the important words via a weighted sum function. Figure 6 presents the structure of our model. The partial description of the incident is "*I cannot access service xxx*". After performing embedding, some values in the output vector will be highlighted by the attention cell (e.g., the fourth value in the output vector).

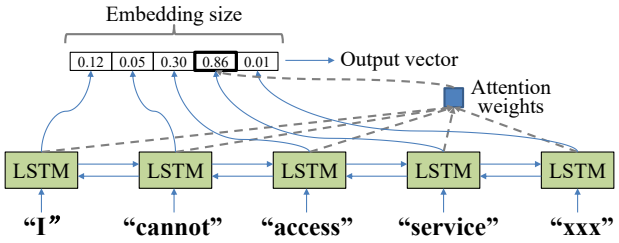

Figure 6: LSTM model for incident embedding.

3.3 Embedding of TSG Description

In this section, we introduce the embedding of TSG descriptions. As introduced before, the TSG description is usually a long-text document. Therefore, a typical RNN model could hardly capture the entire features of documents (also experimentally confirmed in the final evaluation that will be introduced in Section 4.3). As a result, in order to embed TSG descriptions to vectors, we employ a TextCNN model in our approach, which has been evaluated to be effective to handle long texts [30, 49]. The detailed architecture of TextCNN is shown in Figure 7.

TextCNN is a variation of traditional CNN (Convolutional Neural Network) that is widely used in image processing field [31, 43]. The major difference is that TextCNN transforms the two-dimensional convolution for extracting image features into one-dimensional convolution for extracting text features. Through the embedding layer, we can make each word a k-dimensional vector, i.e., $\mathbf{x}_i \in \mathbb{R}^k$ represents the ith word in a sentence. Particularly, we use the vector sequence $\mathbf{x}_{(i:i+n)}$ to represent the concatenation of n words:

$$\mathbf{x}_{(i:i+n)} = \mathbf{x}_i \oplus \mathbf{x}_{(i+1)} \oplus \cdots \oplus \mathbf{x}_{(i+n)} \tag{1}$$

where the operator $\oplus$ denotes the concatenation of vectors. Besides, we define $\mathbf{w}_f \in \mathbb{R}^{hk}$ as a $filter$ convolution operator that can be applied to a sequence of word vectors with a window size of h and produce a new vector. The convolution window moves from the beginning of the sentence to the end. For example, using the $filter$ convolution operator on h consecutive word vectors starting from the ith word outputs the scalar feature c_i:

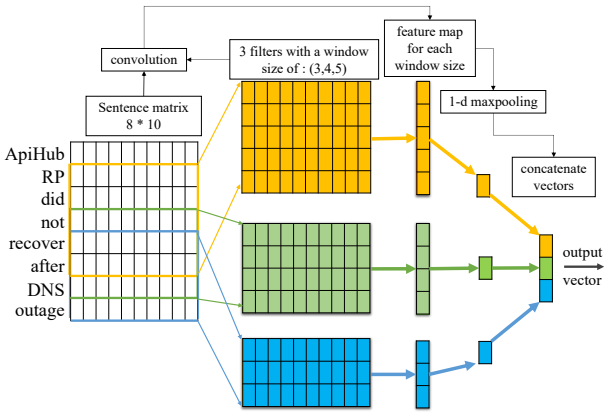

Figure 7: Structure of TextCNN model.

$$c_i = ReLU(\mathbf{w}_f \cdot \mathbf{x}_{(i:i+h-1)} + b_f) \tag{2}$$

where $\mathbf{x}_{(i:i+h-1)}$ is generated from a window of words with the size of h and $b_f \in \mathbb{R}$ is a bias. The symbol "$\cdot$" refers to the dot product and $ReLU(\cdot)$ is the element-wise rectified linear unit function. The $filter$ operator will be applied to each possible window of words in the sentence with n words, i.e., $\mathbf{x}_{(1:h)}, \mathbf{x}_{(2:h+1)}, \cdots, \mathbf{x}_{(n-h+1:n)}$, to produce a feature map $\mathbf{c} \in \mathbb{R}^{(n-h+1)}$.

$$\mathbf{c} = [c_1, c_2, ..., c_{(n-h+1)}] \tag{3}$$

As shown in the figure, we employ three windows (i.e., size of 3, 4 and 5) sliding over the TSG description. That is the new vectors are computed depending on 3, 4 and 5 continuous words respectively. Then, we perform a *maxpooling* operation on each feature map. It is easy to understand that two-dimensional *maxpooling* is to select the maximal value in each fixed-size region of the two-dimensional matrix, and one-dimensional *maxpooling* is to maximize the one-dimensional vector in the two-dimensional matrix. We use one-dimensional *maxpooling* to get a vector consisting of the maximal value of each row. Each different feature map captures different features, and *maxpooling* can choose the one with the highest value (i.e., the most important one) in a feature map. In addition, *maxpooling* can convert variable-length sentences into fixed-length vectors. Finally, we adopt the commonly-used *dropout* technique to further reduce the effects of overfitting, which can make the model more robust and general because it does not heavily rely on local features.

3.4 Joint Learning

According to the previous introduction, we embed both incidents and TSG descriptions into vectors. Then, we employ a joint learning process for model training [16, 21, 46]. Formally, we use the triple $\langle I, T+, T- \rangle$ to denote $T+$ is the desired TSG for incident I while $T-$ is an incorrect TSG for I. In other words, the training input is a set of pair-wised data, which will be used to train a biased selection model, i.e., the computed similarity value between I and $T+$ should

be larger than I and $T-$. In this process, for each incident I, its corresponding $T+$ indicates the correct TSG while $T-$ is a randomly selected one from all other TSGs. In this joint learning process, we use the *cosine* function to measure the similarity. The following is the formula of *cosine* function.

$$cos(\mathbf{v}_I, \mathbf{v}_T) = \frac{\mathbf{v}_I^T \cdot \mathbf{v}_T}{\| \mathbf{v}_I \| \| \mathbf{v}_T \|} \tag{4}$$

In the function, $\mathbf{v}_I$ and $\mathbf{v}_T$ denote the embedding vectors for incident and TSG descriptions, respectively. Finally, we use the following ranking loss function as the optimization target, which is commonly used by previous studies [12, 17, 21].

$$\mathcal{L}(\theta) = \sum_{\langle I,T+,T-\rangle \in D} max(0, \epsilon - cos(\mathbf{v}_I, \mathbf{v}_{T+}) + cos(\mathbf{v}_I, \mathbf{v}_{T-}))$$

In the formula, for each incident I we build the corresponding triplet $\langle I, T+, T-\rangle$, which forms the complete training dataset D and will be used to train the model by minimizing the loss over the dataset D.

4 EVALUATION

In this section, we evaluate the effectiveness of our approach on real-world online service systems. Since there is no prior approach on this topic, we adapt two distinct types of existing approaches for comparison. The first one is a traditional information retrieval (IR) approach utilizing TF-IDF algorithm, which is widely used to measure textual similarities [5, 39]. The second approach is adapted from a recent research proposed by Gu et al. [21] (called DeepCS), which was originally designed for code search. In other words, the "answer" of DeepCS is a code snippet while ours is a TSG description. DeepCS performs code embedding from three perspectives – method names, API sequence and tokens, and finally concatenates them together. However, in our application, we only consider the token sequence of TSG descriptions. To enable comparison, we use a LSTM model to embed the TSG descriptions and use it to replace the code embedding model in DeepCS. Next, we will introduce the details of our experiments.

4.1 Experimental Setup

4.1.1 Subjects. In the evaluation, we use the same dataset that is used in our empirical study. It consists of 18 online service systems and covers all the incident reports in the last two years. Additionally, as presented in the previous study, a portion of TSGs can be handled by existing systems. Therefore, to ensure the proposed approach is useful in practice, we rule out this part of data and only focus on those TSGs that need manual effort to find, which leaves us around 10GB data and 300 kinds of TSGs for the evaluation. Due to sensitivity, we cannot disclose data about the exact number of incident reports and associated time-related information.

4.1.2 Metrics. We employ the metric of *SuccRate@k* to evaluate the effectiveness of our approach, which is commonly used by previous researches [21, 32, 36]. *SuccRate@k* is also known as success percentage at Top-*k* [32], i.e., the percentage of instances that are correctly recommended within Top-*k* positions. Additionally, in our application scenario, the related TSG is unique for each incident. Therefore, for a recommended TSG, it is either correct or not. Since developers need to confirm the correctness of the recommended TSG, too many recommendations will not help much as it may cost more time of developers to check one by one. Especially, we consider *SuccRate@k*, where $k \in \{1, 3, 5\}$ in our evaluation.

4.1.3 Implementation. We build our model with the Pytorch framework. For incident description embedding, we configured the hyper parameters of LSTM model with *dropout* as 0.25, *learning rate* as 0.0001. For TSG description embedding, we set the *window size* of TextCNN model as [3, 4, 5]. Finally, the embedding size is 300. Besides, since the lengths of descriptions in TSGs vary and may involve irrelevant text, we only utilize the first 10 lines of non-empty text by default. For other configurations of DeepCS, we adopt the default configuration in the original paper.

Additionally, our application scenario is time sensitive, where the training data should be produced prior to the testing data. Therefore, we divide our dataset into two distinct subsets as training and testing data respectively according to the chronological order of incident creation. More concretely, we use the first 80% data for training while the remaining 20% for testing, which ensures the accessibility of training data for predicting.

4.2 Research Questions

In our evaluation, we answer the following research questions.

- How effective is DeepRmd in recommending TSGs?
- How effective is each component of DeepRmd?
- How dose the size of TSG description affect the prediction results?

4.3 Experimental Results

4.3.1 Overall Effectiveness of DeepRmd. As introduced in previous sections, we evaluate our approach and compare it with the other two baseline techniques, i.e., traditional IR-based approach that utilizes TF-IDF algorithm and the adapted DeepCS approach. Table 1 presents the experimental results of both DeepRmd and the baseline approaches, where the first column denotes the corresponding approaches, and the following columns show the results on *SuccRate@k*. DeepRmd is able to correctly predict 80.3% of TSGs at top 1, significantly outperforming the two baseline techniques. Particularly, the improvements are from 47.6% to 104.8% with respect to *SuccRate@1*. Please note that, this metric (*SuccRate@1*) is more important than the others as it denotes that when an incident comes, the corresponding mitigation process can be correctly recommended and reused without any delay since the prediction process can be finished within about dozens of milliseconds. It has the potential to greatly speed up the mitigation process. Additionally, if we consider other ranks of correct recommendations (i.e., *SuccRate@3,5*), our approach is still superior to the baseline techniques, where the improvements range from 26.9% to 115.7%. Particularly, if checking the top 5 recommendations, our approach can recommend the desired TSGs for about 94.7% incidents.

The competitive results of our approach can be attributed to the two important components, i.e., the TextCNN and the attention mechanism. Especially, the attention mechanism greatly contributes to the results. For example, for the incident description of *"[AX]*

Table 1: Evaluation results of DeepRmd comparing with baseline.

	SuccRate@1	*SuccRate@3*	*SuccRate@5*
TF-IDF	39.2%	42.6%	46.7%
DeepCS	54.4%	65.8%	74.6%
DeepRmd	80.3%	91.9%	94.7%

- Watch Dog RuleName DatabaseSpaceUsedRule 10PercentRemaining for Tenant 9a083aab-e8d6-459d-8407-xxxxx ..." shown in Figure 1, with the attention mechanism, the words "watch", "rule" and "database" are correctly identified and highlighted (i.e., with bigger weights). As a result, they exactly match those in the TSG description which looks like *"Watchdog Rule Failure: Database Space Used ...'*. However, without the help of the attention mechanism, the network will fail to identify those keywords and miss the correct recommendation. On the contrary, for traditional IR-based technique, the words such as "watch", "rule" and "database" are frequently appeared in different TSGs and TF-IDF cannot capture the contextual information of these words, and thus it fails to give the accurate recommendation.

In summary, the experimental results indicate that our approach is effective and outperforms the baseline approaches.

4.3.2 Effectiveness of Each Component. As introduced in the previous section, compared with the commonly-used text query techniques [16, 21], our approach has two major improvements in model design. First, we adopt the attention mechanism when embedding the incident descriptions. Second, we employ the TextCNN model for TSG description embedding. To evaluate the effectiveness of each component in DeepRmd and explore different combinations of models, we conduct an extensive experiment on a set of variants of DeepRmd with different configurations. The details are listed as follows.

DeepRmd$_{natt}$ In this variant, we remove the attention mechanism for incident embedding and keep others unchanged, aiming at exploring the impact of attention on the results.

DeepRmd$_{lstm}$ This variant changes the TextCNN model to LSTM model. In other words, the TSG description embedding process shares the same model with that of incident embedding. Through this variant, we aim to investigate the effectiveness of TextCNN in DeepRmd.

DeepRmd$_{cnn}$ Similar to DeepRmd$_{lstm}$, this variant changes the incident embedding model to a TextCNN model. That is to say that both two embedding processes are using the TextCNN. The target is to validate the effectiveness of LSTM model for incident embedding.

DeepRmd$_{inv}$ The above variants are exploring the effectiveness of individual components. For this variant, we simply inverse the two embedding models for incident and TSG description in DeepRmd to explore whether the current combination is a relatively better choice.

DeepRmd$_{fc}$ As shown in Figure 5, we employ the simple *cosine* function to compute the similarity between embedded vectors. In order to investigate its effectiveness and to see whether a more complex neural network brings better results in the experiment, we replace it with a full-connected network to measure the similarity.

To compare the effectiveness of different variants, we repeat our experiments multiple times. Except the differences introduced above, all the other experimental settings are the same. We present the final results in Table 2.

Table 2: Evaluation results of different variations

	SuccRate@1	*SuccRate@3*	*SuccRate@5*
DeepRmd$_{natt}$	69.2%	90.3%	94.4%
DeepRmd$_{lstm}$	71.0%	89.8%	93.3%
DeepRmd$_{cnn}$	71.2%	86.6%	86.9%
DeepRmd$_{inv}$	70.1%	90.9%	95.4%
DeepRmd$_{fc}$	27.6%	53.4%	66.5%
DeepRmd	80.3%	91.9%	94.7%

According to the experimental results shown in the table, our approach achieves the best result among all variants with respect to both *SuccRate@1* and *SuccRate@3*. Particularly, except for DeepRmd$_{fc}$ all other variants achieve similar results, demonstrating the effectiveness of the *cosine* function. Additionally, from the table we can further confirm that the attention mechanism is important as it causes the largest impact on the final results, i.e., DeepRmd$_{natt}$ achieves the lowest success rate on top 1. From the table, we can find that when using the same model to embed incident descriptions and TSGs, the results of DeepRmd$_{lstm}$ are similar to those of DeepRmd$_{cnn}$, which only employ either LSTM with attention or TextCNN model. However, both of them are much less effective than our approach, indicating that TextCNN is effective in handling long texts (i.e., TSGs) while LSTM should be a more appropriate method for short texts (i.e., incident descriptions). Overall, our approach DeepRmd achieves better success rates than the variants, and the improvements range from 12.8% to 190.9% in terms of *SuccRate@1*.

In summary, our approach performs better than the variants of different configurations. Especially, when considering *SuccRate@1*, DeepRmd achieves the best result, which is preferable as it can lead to tangible benefits of reducing the mitigation time for incidents.

4.3.3 Impact of Employed TSG Description Size. In our experiment, we read the first 10 lines of TSG descriptions by default. To explore the impact of different sizes of used texts, We further conduct a set of comparison experiments. We utilize different sizes of TSG descriptions and feed them to the TextCNN model of DeepRmd while keep other configurations unchanged. Table 3 reports the details of the experimental settings and the corresponding results.

In the table, the first column denotes the lines of TSG text used in the experiments. From the table, we can find that the impact of TSG text size on the final results is small, especially when the number of lines is less than 20. The results demonstrate that the TextCNN model is not sensitive to the length of input text, which can be attributed to its convolutional structure as it does not face the long-distance dependency problem. When reading the first 10 lines of TSG descriptions, DeepRmd performs the best with respect to *SuccRate@1*. However, when the entire TSG text is used the result becomes worse. The reason is that TSGs record the complete

Table 3: Evaluation results when reading different lines of TSG description from the beginning.

Line Number	*SuccRate@1*	*SuccRate@3*	*SuccRate@5*
1	78.5%	90.8%	93.3%
3	79.5%	91.2%	94.0%
5	79.8%	92.0%	94.7%
10	80.3%	91.9%	94.7%
20	79.9%	91.4%	94.8%
50	75.6%	90.7%	94.4%
All	73.9%	92.8%	94.4%

mitigation process of incidents and thus contain many irrelevant texts, such as database queries for symptom investigation, such as the example shown in Figure 1. As a result, when using the complete description, more noise would be introduced.

5 LESSONS LEARNED

Recommendation Interpretability. Since the TSGs are recommended to developers, who will check their compatibility before applying them, therefore, the interpretability of the recommendation will affect the time cost of manual examination, and thus further affect the time for mitigation, especially for TSGs with a long description. As shown in Table 1, a small portion of incidents are still cannot be accurately recommended by DeepRmd. To dealing with this issue, the attention mechanism in DeepRmd will aid in some degree, which highlights the critical and representative words in an incident by assigning a larger weight. Therefore, when the TSG is too long, checking the vector representation of the incident potentially can help reduce the time cost. However, a user-friendly way to present such information to developers still needs further investigation.

Online Refinement. In practice, both incidents and TSGs may evolve, such as changing the descriptions of incidents and introducing new TSGs, etc. In these cases, the learned model can produce unsatisfiable recommendations and reduce the usability of the approach. To tackle such issues, incremental learning [45, 47] and active learning [40, 41] can be further incorporated, where the former can reuse the historical learned knowledge with small training overhead for timely recommendation while the latter can continuously refine the model via incorporating developers' feedback to the model training process.

6 THREATS TO VALIDITY

Threats to internal validity relate to factors that affect the claims and results in the paper. There are mainly two such factors in this work. The first one is about the experimental settings. We employ two distinct existing techniques that are related to our approach as baselines for comparison, which are either commonly-used (i.e., TF-IDF) or recently proposed (i.e., DeepCS). Besides, to mitigate the implementation bias, we have used the open-source implementation of DeepCS and adopt its default configuration in our evaluation. In addition to the comparison with baselines, we also systematically explored the impact of different configurations of our model. The second factor is about the implementation of our approach. To avoid implementation errors, we have employed a mature and widely-used deep learning framework – Pytorch.

Threats to external validity refer to the generalizibility of our approach. The main factor is the dataset utilized in our empirical study and evaluation, which is from the online services of Microsoft. It may not be generalizable to a wider range of online services from other companies. To mitigate the threat of using a single system, we have used 18 distinct online service systems, some of which are widely used products around the world. Moreover, we have used all data from the past two years for these systems, which cover different application domains, such as cloud (e.g., Microsoft Azure), development (e.g., Visual Studio), and social communication (e.g., Skype). We believe that these large-scale systems could be representative to some extent.

Threats to construct validity mainly refer to the inaccuracy of the measurements in our empirical study. In the study, we rely on the historical incident records, which may not be accurate. For example, the percentage of incidents shown in Figure 2 would be affected if the developers had used a TSG during the mitigation process but forgot to record it. In other words, the percentage of incidents that use TSGs could be higher. As a result, the average mitigation time comparison of incidents would be affected as well (*ref.* Figure 3). Finally, we have used the timestamp of attaching a TSG to the incident to approximate the time to locate the TSG. The result may not be very accurate since the developers may not record the TSGs as soon as they find them, i.e., the recorded time could be later than that in reality. Similar threat affects the entire mitigation time as well. As a consequence, we have employed different service systems developed by hundreds of teams, aiming to mitigate the inaccuracy caused by a small number of developers.

7 RELATED WORK

7.1 Incident Management

As online service systems become more and more popular, incidents are almost inevitable. Recently, researchers have noticed this issue and conducted a series of investigations. For example, Lou et al. [34, 35, 48] carried out a two-year research on large-scale, real-world online systems and reported their experience on the management and diagnosis of incidents. Similarly, Karim et al. [27] reported their lessons learned from the study on issue management and reported a set of insights to help product managers. Additionally, Chen et al. [8] explored the prevalence of reassignment during incident triage process in online service systems. Besides, they empirically investigated the applicability of traditional bug-triage techniques to incident triage, which aims to provide useful insights for future study. Recently researchers also proposed various techniques to assist manual management of online services. For example, Kikuchi [29] proposed to predict the workload of resolving incidents in incident management, which leverages text mining techniques for updating histories of incident tickets. Chen et al. [9] proposed DeepCT, which incorporates a GRU-based model with an attention-based mask strategy to perform continuous incident triage. It can incrementally learn knowledge from discussions and update incident-triage results. Cohen et al. [11] employed probabilistic models (Tree-Augmented Bayesian Networks) to perform automated performance diagnosis and management on Internet

server platforms. Similarly, Duan and Babu [15] employed an active learning algorithm to make the best use of manual effort and help the failure diagnosis process. In this paper, we conduct the first empirical study on the usage and characteristics of troubleshooting guide in online service systems, which complements the previous research. Another related research was conducted by Lim et al. [33], which proposed to use Hidden Markov Random Field for performance issue clustering, targeting to identify repetitive issues to speed up the troubleshooting process. However, it is different from our work as we aim at the recommendation of troubleshooting guides but not the classification of performance issues.

7.2 Bug Report Management

There are also a lot of research on bug reports for software systems. For example, Zhou et al. [52] proposed BugLocator, which locates candidate buggy source files based on the textual similarity between bug reports and source code with considering previous similar bug information. There are also many other approaches that use bug report to assist in fault localization. For example, Huang et al. [25] leveraged ensemble learning technique to recommend the affected package of source code based on the description of bug reports. Fujiwara [18] proposed to recommend programmers a bug report that is likely to contain failure descriptions related to a source file being inspected. Similar to incidents, bug reports have the associated responsible developers as well. Much research has been conducted to automatically assign bug reports to developers. For example, Hu et al. [24] proposed BugFixer, which leverages the historical bug-fix information for developer recommendation when new bug reports come. Jalbert and Weimer [26] and Tian et al. [44] proposed approaches aiming to identify duplicate bug reports and thus reduce human efforts of manual tagging. As discussed in [8], incident reports and bug reports are similar but still different. Traditional bug reports are written manually by submitters, while most of the incident reports for online service systems are created and submitted automatically by monitors of the systems. Also, for an online service system, incidents occur more frequently than bugs as a variety of factors can lead to incidents. Because incidents affect service availability and cause financial loss, timely mitigation of incidents is critical.

8 CONCLUSION

Recently, online services are becoming increasingly popular. Because they provide 7x24 availability to users around the world, incidents are inevitable and may cause great financial loss. Therefore, mitigating incidents is important and time critical. In Microsoft, a mitigation process is usually recorded in a document, called a *troubleshooting guide* (*TSG*). To explore the usage and characteristics of TSGs in real-world online services, we conducted the first large-scale empirical study on 18 online services in Microsoft. According to the empirical result, we proposed a TSG recommendation technique, DeepRmd, to facilitate the incident mitigation process. DeepRmd leverages deep learning techniques to identify the textual similarity between incident description and its corresponding TSG, based on which it recommends a list of candidate TSGs that may guide the developer to mitigate a given incident. We have evaluated the effectiveness of DeepRmd on real-world online service systems. The result indicates that DeepRmd can recommend the correct TSG for around 80.3% incidents as the Top 1 returned result, which significantly outperforms the baseline approaches.

ACKNOWLEDGMENTS

This work was partially supported by the National Key Research and Development Program of China under Grant No.2017YFB1001803, the National Natural Science Foundation of China under project No.61702107, and Australian Research Council (ARC) DP200102940.

REFERENCES

[1] 2016. Cost of Data Center Outages. https://www.vertiv.com/globalassets/documents/reports/2016-cost-of-data-center-outages-11-11_51190_1.pdf.
[2] 2018. Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales. https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7.
[3] 2019. ANNOY library. https://github.com/spotify/annoy. Accessed: 2019-09-01.
[4] Aug., 2008. Amazon's S3 cloud service turns into a puff of smoke. InformationWeek.
[5] Akiko Aizawa. 2003. An information-theoretic perspective of tf-idf measures. *Information Processing & Management* 39 (2003), 45 – 65. https://doi.org/10.1016/S0306-4573(02)00021-3.
[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *ICLR*. http://arxiv.org/abs/1409.0473.
[7] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5, 2 (1994), 157–166. https://doi.org/10.1109/72.279181.
[8] Junjie Chen, Xiaoting He, Qingwei Lin, Yong Xu, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. An Empirical Investigation of Incident Triage for Online Service Systems. In *ICSE-SEIP*. IEEE Press, 111–120. https://doi.org/10.1109/ICSE-SEIP.2019.00020
[9] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. Continuous Incident Triage for Large-Scale Online Service Systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 364–375. https://doi.org/10.1109/ASE.2019.00042
[10] Junjie Chen, Shu Zhang, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Yu Kang, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2020. How Incidental are the Incidents? Characterizing and Prioritizing Incidents for Large-Scale Online Service Systems. In *The 35th IEEE/ACM International Conference on Automated Software Engineering*. to appear.
[11] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*. USENIX Association, 16–16.
[12] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. *J. Mach. Learn. Res.* (Nov. 2011), 2493–2537.
[13] Sanjoy Dasgupta and Yoav Freund. 2008. Random Projection Trees and Low Dimensional Manifolds. In *STOC*. ACM, 537–546. https://doi.org/10.1145/1374376.1374452.
[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* (2018). https://arxiv.org/abs/1810.04805.
[15] S. Duan and S. Babu. 2008. Guided Problem Diagnosis through Active Learning. In *2008 International Conference on Autonomic Computing*. 45–54. https://doi.org/10.1109/ICAC.2008.28.
[16] Minwei Feng, Bing Xiang, Michael R. Glass, Lidan Wang, and Bowen Zhou. 2015. Applying Deep Learning to Answer Selection: A Study and An Open Task. *CoRR* abs/1508.01585 (2015). http://arxiv.org/abs/1508.01585.
[17] Andrea Frome, Greg S Corrado, Jon Shlens, Samy Bengio, Jeff Dean, Marc'Aurelio Ranzato, and Tomas Mikolov. 2013. Devise: A deep visual-semantic embedding model. In *Advances in neural information processing systems*. 2121–2129.
[18] S. Fujiwara, H. Hata, A. Monden, and K. Matsumoto. 2015. Bug report recommendation for code inspection. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. 9–12. https://doi.org/10.1109/SWAN.2015.7070481.
[19] Alex Graves, Abdel rahman Mohamed, and Geoffrey E. Hinton. 2013. Speech recognition with deep recurrent neural networks. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), 6645–6649.
[20] A. Graves and J. Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM networks. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Vol. 4. 2047–2052 vol. 4. https://doi.org/10.1109/IJCNN.2005.1556215.

[21] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *ICSE*. 933–944. https://doi.org/10.1145/3180155.3180167.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735.

[23] J. Nicholas Hoover. Aug. 16, 2008. Outages Force Cloud Computing Users To Rethink Tactics. InformationWeek. https://www.informationweek.com/cloud/software-as-a-service/outages-force-cloud-computing-users-to-rethink-tactics/d/d-id/1071014.

[24] H. Hu, H. Zhang, J. Xuan, and W. Sun. 2014. Effective Bug Triage Based on Historical Bug-Fix Information. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 122–132. https://doi.org/10.1109/ISSRE.2014.17.

[25] Q. Huang, D. Lo, X. Xia, Q. Wang, and S. Li. 2017. Which Packages Would be Affected by This Bug Report?. In *ISSRE*. 124–135.

[26] N. Jalbert and W. Weimer. 2008. Automated duplicate detection for bug tracking systems. In *DSN*. 52–61. https://doi.org/10.1109/DSN.2008.4630070.

[27] M. R. Karim, S. M. D. A. Alam, S. J. Kabeer, G. Ruhe, B. Baluta, and S. Mahmud. 2016. Applying Data Analytics towards Optimized Issue Management: An Industrial Case Study. In *CESI*. 7–13. https://doi.org/10.1109/CESI.2016.012.

[28] Andrej Karpathy and Li Fei-Fei. 2015. Deep visual-semantic alignments for generating image descriptions. In *CVPR*. 3128–3137.

[29] S. Kikuchi. 2015. Prediction of Workloads in Incident Management Based on Incident Ticket Updating History. In *UCC*. 333–340. https://doi.org/10.1109/UCC.2015.53.

[30] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. *CoRR* (2014). arXiv:1408.5882 https://arxiv.org/abs/1408.5882.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60 (2017), 84–90. https://doi.org/10.1145/3065386.

[32] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. 2016. Relationship-aware Code Search for JavaScript Frameworks. In *FSE*. ACM, 690–701. https://doi.org/10.1145/2950290.2950341.

[33] M. Lim, J. Lou, H. Zhang, Q. Fu, A. B. J. Teoh, Q. Lin, R. Ding, and D. Zhang. 2014. Identifying Recurrent and Unknown Performance Issues. In *2014 IEEE International Conference on Data Mining*. 320–329. https://doi.org/10.1109/ICDM.2014.96.

[34] Jian-Guang Lou, Qingwei Lin, Rui Ding, Qiang Fu, Dongmei Zhang, and Tao Xie. 2013. Software Analytics for Incident Management of Online Services: An Experience Report. In *ASE*. 475–485. https://doi.org/10.1109/ASE.2013.6693105 https://doi.org/10.1109/ASE.2013.6693105.

[35] Jian-Guang Lou, Qingwei Lin, Rui Ding, Qiang Fu, Dongmei Zhang, and Tao Xie. 2017. Experience report on applying software analytics in incident management of online service. *ASE* (2017), 905–941. https://doi.org/10.1007/s10515-017-0218-1 https://doi.org/10.1007/s10515-017-0218-1.

[36] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *ASE*. 260–270. https://doi.org/10.1109/ASE.2015.42.

[37] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model.. In *Interspeech*, Vol. 2. 3.

[38] Anand Rajaraman and Jeffrey David Ullman. 2011. *Data Mining*. Cambridge University Press, 1–17. https://doi.org/10.1017/CBO9781139058452.002.

[39] Juan Enrique Ramos. 2003. Using TF-IDF to Determine Word Relevance in Document Queries.

[40] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. 2020. A Survey of Deep Active Learning. arXiv:2009.00236 [cs.LG]

[41] Christopher Schröder and Andreas Niekler. 2020. A Survey of Active Learning for Text Classification using Deep Neural Networks. arXiv:2008.07267 [cs.CL]

[42] Mike Schuster and Kuldip Paliwal. 1997. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on* 45 (12 1997), 2673 – 2681. https://doi.org/10.1109/78.650093.

[43] Vishwanath A. Sindagi and Vishal M. Patel. 2018. A survey of recent advances in CNN-based single image crowd counting and density estimation. *Pattern Recognition Letters* 107 (2018), 3 – 16. https://doi.org/10.1016/j.patrec.2017.07.007.

[44] Y. Tian, C. Sun, and D. Lo. 2012. Improved Duplicate Bug Report Identification. In *2012 16th European Conference on Software Maintenance and Reengineering*. 385–390. https://doi.org/10.1109/CSMR.2012.48.

[45] Yue Wu, Yinpeng Chen, Lijuan Wang, Yuancheng Ye, Zicheng Liu, Yandong Guo, and Yun Fu. 2019. Large Scale Incremental Learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 374–382.

[46] Ran Xu, Caiming Xiong, Wei Chen, and Jason J Corso. 2015. Jointly modeling deep video and compositional text to bridge vision and language in a unified framework. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[47] Yang Yang, Da-Wei Zhou, De-Chuan Zhan, Hui Xiong, and Yuan Jiang. 2019. Adaptive Deep Models for Incremental Learning: Considering Capacity Scalability and Sustainability. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '19)*. Association for Computing Machinery, 74–82. https://doi.org/10.1145/3292500.3330865

[48] D. Zhang, S. Han, Y. Dang, J. Lou, H. Zhang, and T. Xie. 2013. Software Analytics in Practice. *IEEE Software* 30, 5 (Sep. 2013), 30–37. https://doi.org/10.1109/MS.2013.94.

[49] Ye Zhang and Byron C. Wallace. 2015. A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification. *CoRR* (2015). arXiv:1510.03820 https://arxiv.org/abs/1510.03820.

[50] Nengwen Zhao, Junjie Chen, Xiao Peng, Honglin Wang, Xinya Wu, Yuanzong Zhang, Zikai Chen, Xiangzhong Zheng, Xiaohui Nie, Gang Wang, Yong Wu, Fang Zhou, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2020. Understanding and Handling Alert Storm for Online Service Systems. In *The 42nd International Conference on Software Engineering, SEIP track*. to appear.

[51] Nengwen Zhao, Junjie Chen, Zhou Wang, Xiao Peng, Gang Wang, Yong Wu, Fang Zhou, Zhen Feng, Xiaohui Nie, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2020. Real-time Incident Prediction for Online Service Systems. In *The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. to appear.

[52] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports. In *ICSE*. IEEE Press, 14–24. https://doi.org/10.1109/ICSE.2012.6227210.

[53] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. 2019. Delta Debugging Microservice Systems with Parallel Optimization. *IEEE Transactions on Services Computing* (2019), 1–1. https://doi.org/10.1109/TSC.2019.2919823.

[54] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking Microservice Systems for Software Engineering Research. In *ICSE: Companion Proceeedings*. ACM, 323–324. https://doi.org/10.1145/3183440.3194991.