

Intelligent code reviews using deep learning

Anshul Gupta
Microsoft Corp.
ansgupta@microsoft.com

Neel Sundaresan
Microsoft Corp.
neels@microsoft.com

ABSTRACT

Peer code review is a best practice in Software Engineering where source code is reviewed manually by one or more peers(reviewers) of the code author. It is widely acceptable both in industry and open-source software (OSS) systems as a process for early detection and reduction of software defects. A larger chunk of reviews given during peer reviews are related to common issues such as coding style, documentations, and best practices. This makes the code review process less effective as reviewers focus less on finding important defects. Hence, there is a need to automatically find such common issues and help reviewers perform focused code reviews. Some of this is solved by rule based systems called linters but they are rigid and needs a lot of manual effort to adapt them for a new issue.

In this work, we present an automatic, flexible, and adaptive code analysis system called DeepCodeReviewer (DCR). DCR learns how to recommend code reviews related to common issues using historical peer reviews and deep learning. DCR uses deep learning to learn review relevance to a code snippet and recommend the right review from a repository of common reviews. DCR is trained on historical peer reviews available from internal code repositories at Microsoft. Experiments demonstrate strong performance of developed deep learning model in classifying relevant and non-relevant reviews w.r.t to a code snippet, and ranking reviews given a code snippet. We have also evaluated DCR recommendations using a user study and survey. The results of our user study show good acceptance rate and answers of our survey questions are strongly correlated with our system's goal of making code reviews focused on finding defects.

CCS CONCEPTS

• **Machine learning** → Supervised learning; *Neural networks* •

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

KDD'18 Deep Learning Day, August 2018, London, UK

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

Software and its Engineering → Software functional properties

KEYWORDS

Deep neural networks, automatic peer reviews, code analysis, natural language processing, data mining, recommendation

1 INTRODUCTION

Peer code review [1, 2] is a critical part of software development lifecycle where source code is reviewed manually by one or more peers of the code author. The source code is inspected to discover errors, to ensure that the source code complies with best practice standards, and to discover vulnerabilities such as race conditions, malware, memory leaks, buffer overflows, and format string exploits. Code review is used to find these problems which may have been overlooked by a developer before the software is released.

Parts of this process are being automated by systems called code linters/static analysis tools¹. Linters are made up of rules related to coding best practices. They get triggered whenever these best practices are violated, mostly during the build time in IDEs. The rules used by linters work well for only a few variations of a violation [3] and needs a lot of manual customization to adapt them for specific projects, teams, or , industry. One needs to maintain right set of rules that suits one's projects and add new analyse rules for modifications in best practices. This rigidness of static analysis tools makes them hard to use [4] and unscalable for large and diverse code repositories present in open source software (OSS), and industry.

Even in presence of these systems, peer or code reviews result in common issues related to documentation, style, and, structure. Mäntylä and Lassenius [5] showed that 75 % of code review comments are related to such common issues. Bacchelli and Bird [6] performed an empirical analysis to see the motivation behind the code review performed by reviewers. They found that even with strong motivation to finding defect (bugs or functional issues in code), reviewers tend to focus on reviews which are mostly related to obvious bugs, or best practices. This makes code reviews less effective

¹ https://en.wikipedia.org/wiki/Static_program_analysis

because reviewers are distracted by common issues, and they focus less on important aspects such as defects, architectural issues, and testing for code.

To address challenges mentioned above, we developed an adaptive, scalable, and easy to configure automatic code analysis system called DeepCodeReviewer (DCR). The system can automatically review code for common issues so that a developer can proactively correct the code, and reviewers can perform code reviews focused towards finding defects. DCR makes use of deep neural networks to automatically learn code analyses corresponding to the human reviews given to code snippets in historical peer reviews. It uses Long Short Term Memory (LSTM) networks [7] to understand source code snippets and reviews, and learns to apply relevant review to a code snippet. This makes our system: 1. *Flexible*, as its code analyses can generalize to new variations of an issue; 2. *Easy to configure*, as it learns to prioritize its analyses from a project or organization's historical code reviews; and 3. *Adaptive*, as it just needs historical code reviews for a specific coding language.

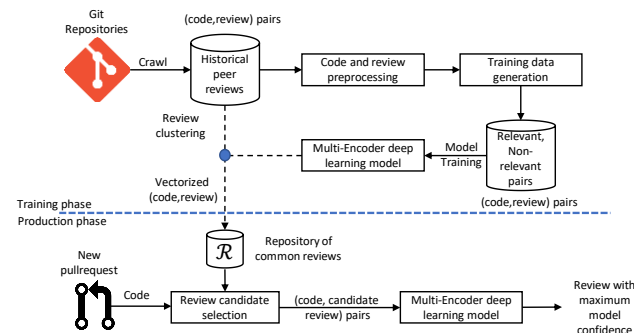


Figure 1 Overall architecture of DeepCodeReviewer with two main phases: training phase processes historical peer reviews and trains a deep learning code review model; production phase makes use of trained model to perform code analysis and apply relevant review.

Figure 1 describes DCR in action. For a new code snippet, DCR finds most relevant review/s from a repository of common reviews generated using historical peer reviews. For scalability, it first finds a set of candidate reviews from repository and then it makes use of a Deep learning model to find most relevant review/s for a code snippet. For this work, we trained DCR for peer reviews in C# language projects. We collected historical data from internal corporate git repositories. In the following section, we present related work for automatic code reviews, and background of code reviews using Git PRs². Then we will explain in detail all the components of DCR. We will then discuss the evaluation of our proposed deep learning model, and results from user study to

evaluate overall performance of DCR. Finally, we will conclude with discussions and future direction for this work.

2 BACKGROUND AND RELATED WORK

2.1 Background on peer reviews

Peer or code review is considered a highly recommended practice in software engineering [8]. It is widely acceptable both in industry and open-source software (OSS) systems as a process for early detection and reduction of software defects. Fagan [9] formalized a highly structured process for code reviewing based on line-by-line group reviews, done in extended meetings for code inspection. This was effective but was time consuming and a heavyweight process. In recent years organizations and OSS use a lightweight code review process with less formal requirements along with supporting tools to perform code review. Some examples of tools used in industry are Google's Mondrian [10], Facebook's Phabricator[11] and the open-source Gerrit³ initiative.

Our code review system is based on Pull requests (PRs) and are very similar to Github's PR review system⁴. A PR asks collaborators⁵ of a project to review or vote code changes associated with a bug fix or a new feature request in a software product. Collaborators then review the submitted code change to make sure that it is free of defects. If they find any, they leave a comment explaining the defect that they have found. The creator of the PR then adds follow-up code changes related to the review comment. Once all the collaborators and the creator are happy with the changes, the PR is then merged to the master code repository or mainline. To train DCR, we used historical PRs from our internal code repository and management system.

2.2 Related work

Most of the work related to automating the process of code review is centered around static analysis of code[12]. Static analysis tools (commonly called linters) aims to examine the text of a program statically, without attempting to execute it. They commonly work on source code representations i.e. abstract syntax tree, or some form of object code. These analyses are shown in IDEs and usually consist of suggestions related to design, security, performance improvement, style, and code readability. They make use of rules and allow end user to add any specific rules related to his/her project or team guidelines. For C#, Microsoft .NET Framework has two static analysis tools: 1. FxCop⁶; 2. StyleCop⁷.

³ Gerrit: [http://en.wikipedia.org/wiki/Gerrit_\(software\)](http://en.wikipedia.org/wiki/Gerrit_(software))

⁴ <https://github.com>

⁵ Collaborators are developers working on a project. One or more collaborators are usually added to a PR as reviewers.

⁶ [https://msdn.microsoft.com/en-us/library/bb429476\(v=vs.80\).aspx](https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx)

² <https://help.github.com/articles/about-pull-requests/>

Most of the static analysis is shown in the IDEs, and even in presence of static analysis tools, code reviews tend to focus on the common issues[5, 6]. This prompted the development of tools that can run static analysis tool for each PR and present the analysis centrally to both the developers and reviewers of a project or a team. Balachandran [13] developed a system called ReviewBot that uses the output of multiple static analysis tools to automate checks for coding standards violations and common code patterns, and published the reviews automatically to the code review request. Although they showed high acceptance rate for their suggestions, their system doesn't understand priority of the rules that they configure for a project, and configuration can be different for different projects. A system called Tricoder [4] provides program analysis ecosystem that is scalable and sharable across multiple repositories. It combines many static analysis tools and allows developers to add their own static analyses into the system.

More recently, machine learning based program analysis has become quite popular. Allamanis, et al. [14] present a survey of machine learning techniques used for applications such as code search [15], syntax error correction [16], and detection [17], and bug detection[18]. The basic premise is centered around naturalness hypothesis:

"Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools"

Bielik, et al. [3] presents a machine learning based technique to learn static analysis rules automatically from a given dataset instead of manually providing various inference rules. They trained their system by running existing static analyzers for issues related to learning point-to analysis rules and allocation site rules. They used decision tree algorithm to generate rules, showed that generated rules have better coverage for tricky corner cases than existing analyzers. Although they showed that their system can improve coverage of an existing static analysis system, it cannot adapt to modifications of the same rule for different projects or team, and lacks the capability to find new issues.

With the advent of GPU computing and scalable machine learning methods, deep neural networks have become state-of-the-art methods for solving machine learning problems. Deep neural networks such as Long Short Term Memory (LSTM) networks have been widely used in analyzing sequences of natural language word tokens. They have gained popularity for text generation and are used in machine translation [19], image and video descriptions [20], etc. These application of LSTMs in text are adapted and used for code retrieval [15], summary generation for source code [21], and

code synthesis [22]. Recently, Allamanis, et al. [23] presented a graph based deep neural network technique to find two specific issues in code: variable misuse, and variable naming. Along with sequence of tokens, they embed code context using abstract syntax trees and data flow graphs. They then train their model in an unsupervised way (not directly for the problem of VarMisuse, or VarNaming), and then apply the trained model to predict if there is a mismatch between the variable used and the variable predicted by their system.

3 DeepCodeReviewer (DCR)

Given a code snippet and a repository of reviews, DCR performs review recommendations. For this, we developed a deep learning model that takes as input (code, review) pairs and outputs score that indicates the relevance of each review to its corresponding code pair. Figure 1 shows the overall architecture of DCR. During training phase, we first preprocess (code, review) pairs to remove noise and convert them to sequences. Then we train a deep learning model using generated data of relevant and non-relevant (code, review) pairs. For any new code snippet, we apply the same preprocessing steps to convert it to sequence, and then make use of deep learning model to find most relevant reviews from a repository of common reviews given in historical code reviews. Sections below explain each component of DCR in detail.

3.1 Code and review pre-processing

Each PR consists of code snippets and reviews given on those code snippets. In our work, we want to learn to apply relevant review to a code snippet by analyzing sequence of code and review tokens.

3.1.1 Code tokenization

We tokenize code using a system called Lexer⁸. Lexer is a tokenizer designed specifically to split code statements into lexical units called tokens. For example, for a code statement `x = a + b * 2`, the lexer output is

```
[(identifier, x), (operator, =), (identifier, a), (operator, +),
(identifier, b), (operator, *), (literal, 2), (separator, ;)]
```

where each tuple is made of token name and its values. The token name is a category of the token; categories for C# are shown in Table 1. Once we have these sequences of tuples, we then transform it to generate sequence of token values.

3.1.2 Review tokenization and filtering

Code review comments are like tweets. They have links, '@' annotations with user names, and instead of images they have

⁷ <https://en.wikipedia.org/wiki/StyleCop>

⁸ https://en.wikipedia.org/wiki/Lexical_analysis

code blocks. For this reason, we used nltk's [24] tweet tokenizer to tokenize the review comments.

We filter reviews if they belong to certain types such as praise and affirmations, reviews with code blocks only, and reviews related to typos in variable names. This kind of filtering helps in removing noise from our data and direct the model training towards learning actionable reviews that will lead to a code change. Review filtering is explained in more details in next section.


Table 1 Major token name categories produced by pygment lexer⁹

Token Name	Description
Keyword	Any kind of keywords such as using, namespace, class, public etc.
Name	variable/function names
Literal	Any literals (number and string)
Operator	operators (+, not...)
Punctuation	punctuation ([, (...)
Comment	any kind of comments

3.2 Code and review relevance deep learning model

Our deep learning model takes input pair of **code snippet** (C_i) and review (R_i) where i stand's for i^{th} code snippet and review pair in dataset. Each C_i and R_i is converted into sequence of token $C_i \rightarrow (c_1^i, c_2^i, \dots, c_n^i)$ and $R_i \rightarrow (r_1^i, r_2^i, \dots, r_p^i)$ where n is number of code tokens in a code snippet, and p is the number of review tokens in a review text. We represent set of code sequences as S_C , and set of reviews as S_R .

3.2.1 Representing code and review tokens using word2vec

 Convert both **code and review tokens to an n-dimensional valued vector using word2vec algorithm** [25]. This n-dimensional real valued vector is commonly referred as embedding vector. These vectors are dense representation of tokens and does not have sparsity issues. Word2vec embedding vectors are known for capturing semantic relations between tokens. For example, we will see cluster of embeddings for all code access modifiers tokens, variable name tokens that represents a specific type, and tokens that have occurred in say 'if' statement, for-each, etc.

To pretrain these embedding vectors on set of code, and reviews, we applied Gensim¹⁰ word2vec on S_C and S_R individually to generate two mappings

⁹ Pygment is a lexer software written in Python language. It supports many languages along with C#. For detailed descriptions on token name categories visit <http://pygments.org/docs/tokens/>.

¹⁰ <https://radimrehurek.com/gensim/models/word2vec.html>

$$T_C \in (c_j^i \rightarrow ec_j^i); j \in [0, n) \quad (1)$$

$$T_R \in (r_k^i \rightarrow er_k^i); k \in [0, p) \quad (2)$$

where n is size of code token vocabulary, p is size of review token vocabulary; $ec \in \mathbb{R}^{H_C}$ and $er \in \mathbb{R}^{H_R}$ are embedding vector for code and review token respectively and H_C and H_R are embedding dimensionality.

3.2.2 Training data generation

We train our Deep learning model to solve binary classification problem: given a tuple $\langle C_i, R_i \rangle$ train a model to predict if R_i is relevant to C_i or not. For this, we should have a dataset of relevant (+ve) and non-relevant (-ve) tuples of code snippets and reviews. But the data that we get from historical PRs just consists of +ve pairs. Therefore, we must artificially generate -ve pairs to have a dataset that can be used for training and validation of deep learning model explained in next subsection.

We generate -ve pairs by selectively sampling reviews. We begin with a repository of $\langle \text{code}, \text{review} \rangle$ tuples (+ve pairs) generated by crawling PRs; let's define it as τ . Then for each +ve pair $\langle C_i, R_i \rangle$ we randomly sample 4 reviews without replacement from R_l where $l \in [0, m); l \neq i$, where m is the size of τ . **Then we generate -ves by pairing C_i with each of the 4 sampled reviews.** We repeat this process for each +ve pair in τ .

This process generates -ve pairs that will help our deep learning model to learn to not apply other reviews but only R_i to C_i . But we also want our model to learn to not apply R_i to corrected C_i . For this, we made use of the final version of the code snippet that got merged into the mainline once the PR is completed. This version of code should have corrections related to the review R_i . We then generate an additional -ve pair using the final version of the code snippet and R_i .

3.2.3 Multi-encoder deep learning model

To find if a review is relevant to a code or not, we designed a deep neural network (F) that understands a code snippet in three parts: upper lines, current line, and lower lines; see Figure 2. Current line is the line where the review comment is posted by a reviewer and divided a code snippet into two equal parts. This way of splitting the code helps the model to give importance to parts of code snippet most relevant to the review given.

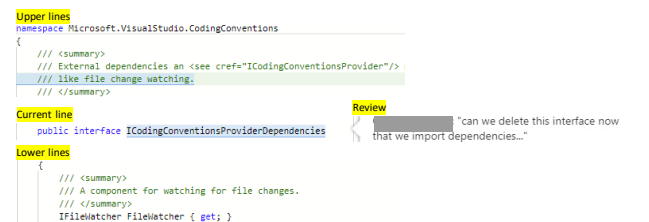


Figure 2 Example of a code snippet and its different parts

Each part of code snippet and the review is converted into sequence of tokens using the tokenizer in section 3.1. On each token, mapping from eq(1) and eq(2) are applied to convert the token to its corresponding embedding vector.

Sometimes review text doesn't contain enough information for the model to learn that a review is applicable to a code or not. In such cases code context (ctx), injected in later stage of F , helps model to learn review's applicability. To extract code context, we align the review comments to the code tokens, and extract unique set of token names occurred in the current line the comment is given, and the few neighboring lines. We convert the unique set tokens to a vector, and this vector becomes our code context (ctx).

We use 4 LSTM models to individually analyze three code parts, and the review R_i . Output of these 4 LSTMs is combined using another feed forward deep neural network whose output layer has a fixed size of two neurons (relevant and non-relevant) with SoftMax activation, which outputs a relevance score ($S_{relevance}$) between [0,1]. Equations below explains the model in details.

$$S_{relevance} = F(C_i^{upper}, C_i^{current}, C_i^{lower}, ctx, R_i) \quad (4)$$

where

$$S_{relevance} \propto Softmax(W_f A) \quad (5)$$

and,

$$A = Relu(W_1[e_{upper}|e_{current}|e_r|ctx]) \quad (6)$$

$$Relu(W_2[e_{lower}|e_{current}|e_r|ctx])$$

where

$$\begin{aligned} e_{upper} &= LSTM_{upper}(T_c[C_i^{upper}]) \\ e_{current} &= LSTM_{current}(T_c[C_i^{current}]) \\ e_{lower} &= LSTM_{lower}(T_c[C_i^{lower}]) \\ e_r &= LSTM_r(T_r[R_i]) \end{aligned} \quad (7)$$

where $|$ is a concatenation operator; $C_i, C_i^{upper}, C_i^{current}$, and C_i^{lower} are sequences of code tokens; R_i is sequence of review tokens; i represent i^{th} code snippet and review pair in τ ; ctx is vectorized version of code context; W_1, W_2 , and W_f are weight matrices for feed forward neural network that combines output of all LSTMs; $e_{upper}, e_{current}, e_{lower}$, and e_r are the encoded representations generated as output of 4 LSTMs; T_c and T_r are the code to embedding mapping functions.

3.2.4 Model training

We perform supervised end-to-end training using back-propagation to tune the parameters of the pretrained embeddings, LSTMs, and the combiner feedforward neural

network. We use Adam optimizer[26] to minimize the cross-entropy objective function on our training set. To prevent overfitting in LSTM layers we use dropout[27] and use L2 regularization in feedforward fully connected layers. We add class weights to force the model to learn from the imbalanced dataset of relevant and non-relevant (code, review) pairs.

3.3 Review recommendation for a new code snippet

When a new PR comes, it can consist of new code files or modifications to some existing code file in a repository. DCR starts with first splitting code files into smaller code snippets. For each code snippet, a set of candidate reviews (C_R) from a repository of reviews (\mathcal{R}) are found. This is done to make our system efficient by avoiding matching all the reviews in \mathcal{R} to a code snippet. Then the code snippet is paired with each candidate review to generate code review tuples, and perform code and review preprocessing. We then pass these tuples as input to the Multi-encoder deep learning model. The model outputs $S_{relevance}$ for each candidate review. Reviews that cross a certain threshold are the DCR's recommendations for a code snippet. This process is repeated for all the code snippets of a file.

3.3.1 Repository of review generation

The most critical part here is to generate \mathcal{R} from historical PR reviews. To recommend diverse reviews, \mathcal{R} should have reviews that belongs to diverse code issues and are also frequently given during historical code reviews - frequent reviews are better learned by model than rare reviews. To get diverse review types, we first perform phrase extraction using Autophrase [28]. Based on phrases, we clustered all the reviews, and labeled clusters by performing visual analysis of the review text in the clusters. More details on the types in present in next section.

In \mathcal{R} we store the raw text of the review, it's encoded version of review, and encoded version of the upper, lower and current code lines of the code snippet where the review was given; see eq(7).

3.3.2 Candidate review selection

To generate candidate set of reviews for a new code snippet, we first split code snippet into upper, lower and current lines. We then use the LSTMs (see eq 7) to get the encoded representation of the three parts of the new code snippet and calculate cosine similarity between the new and all the code snippets in \mathcal{R} (for all the three parts individually). We select the reviews from \mathcal{R} whose code snippets are like the new code snippet in all upper, lower and current parts.

3.3.3 Threshold selection for review recommendation

To find the most applicable reviews to the new code snippet we threshold the $S_{relevance}$ for candidate reviews. To select

threshold, we first calculate $S_{relevance}$ for all the $\langle C_i, R_i \rangle$ tuples in \mathcal{R} . We then calculate large confidence interval for $S_{relevance}$ in each cluster. The threshold for a cluster is chosen as the upper limit of the confidence interval. Upper limit is selected so that reviews for which DCR is highly confident are recommended.

3.4 Dataset

We trained and tested our system on C# code reviews. For this, C# PRs (Pull Requests) are crawled from 208 representative code repositories within our organization. Each PR contains a lot of information, but for our case only modified code, final merged code, and the review text are important. The dataset contains 22,435 completed C# PR and 56,052 code and review comment pairs.

3.4.1 Review filtering

Our goal is to recommend an actionable review for a new code snippet. An **actionable review results into a valid code change**. A deeper analysis of our data revealed that a good portion of human written reviews are not actionable. Here are a couple of examples.

- Reviews asking for explanation or related to discussion on functional part of a code change e.g. "what's the point of this change?", "maybe a good op to fix this error message are you sure * you want * to do this ?", etc.
- Reviews related to affirmation, praise, joke: "omg , what kind of monster would've checked this in the first place ? :-)", "love it :-)", "that was really it :-) amazing".

There are also reviews that require code context that is richer, and distant from the location of the reviews. For instance,

- Reviews related to addition of a test in a test file corresponding to a change e.g. "need unit / regression tests", "do not change old tests unless required report needs to keep working add new tests for new functionality these are just as much regression tests as unit / integration tests".
- Reviews related to business logic, or functional change e.g. "can this be a passthrough as well it also seems like we don't need this entire class at this point", "you still want this part", etc.

We notice that longer reviews tend to be quite specific and do not repeat a lot. Hence, training data won't have enough instances of the same issue and our model won't be able to learn it well. Also, LSTMs tend to remember recently seen tokens in longer reviews and forget tokens from beginning part of the review. These reasons prompted us to select short reviews.

For the non-actionable, and functional reviews, we design **vs. star expressions filters** to detect and remove them from our dataset. After filtering we were left with 30,509 code and review pairs.

3.4.2 Training and validation dataset

As explained in section 3.2.2, we generate a dataset of relevant and non-relevant code and review pairs. We experimented generating training data with 1,3, 5, and 11 -ves for each +ve pair. Using each generated dataset, we trained our model and tested its performance using metrics described in next section. We found model trained on dataset with 5 -ves per 1 +ve gives best performance. There are two reasons why moderate number of negatives worked best: 1. Less number of negatives do not capture the distribution of -ve pairs well; 2. With large number of -ves, there is a higher chance that review sampling for -ve pairs will generate reviews similar to +ve review; model will get confused when trained on such dataset.

We generated 152,545 not relevant -ve pairs using 1:5 ratio for +ves and -ves. We divide this dataset according to 90-10 ratio into train (train_dat) and validation (val_dat) dataset using a stratified sampling method pivoted on the code from the +ve code review pair. All -ve pairs generated for a +ve pair either become part of training or validation dataset. Using stratified sampling ensures that +ve and -ve pairs remains in 1:5 ratio in both training and validation. Table 2 shows details on train_dat, and val_dat.

Table 2 Shows training and validation dataset details

Dataset	Size
Training (train_dat)	164,748 code review pairs 27,458 +ve; 137,290 -ve
Validation (val_dat)	18,306 code review pairs 3051 +ve; 15,255 -ve

3.4.3 Repository of reviews

As explained in section 3.3.1, we generate a repository of reviews (\mathcal{R}) that has diverse and frequently given reviews. Using the process explained, **we found 139 different types of reviews in 30,509 reviews**. As shown in Figure 3, these 139 review types can be clubbed into 7 broad categories. In our system that we used to perform user study (explained in following section).

3.4.4 Model implementation details

We used Keras [29] to implement our code review deep learning model. We used embedding dimensionality of 200 for code and review token and set LSTM encoder hidden state dimensionality to 400. The model is trained for 16 epochs, and after each epoch cross-entropy objective function is computed on training and validation set.

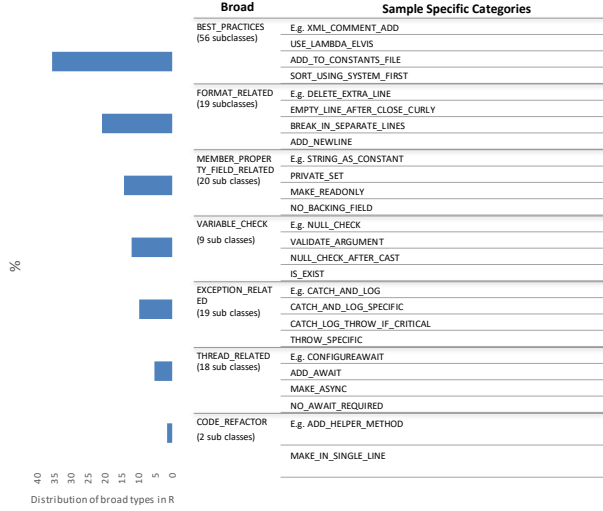


Figure 3 Distribution of reviews in R

4 Evaluation and Results

We evaluate the code review model using automatic metrics and evaluated the overall DCR system using a user study.

4.1 Automatic metrics

4.1.1 Classification metrics for multi encoder code review model (DeepMEM)

We evaluated our deep learning model for its ability to classify relevant and non-relevant (code, review) pairs. For this we measured area under precision-recall (PR) curve on val_dat. We compared our method performance with a baseline method that uses TF-IDF to represent (code, review) pairs, and logistic regression for classification.

In the baseline method, the TF-IDF vectors are first calculated for the code and each of the review comment. Then both the vectors are concatenated to form a single feature vector for a code and review tuple. Then a logistic regression, with L2 regularization and class weights, is trained using the concatenated vectors to classify pairs as related or not-related.

Figure 4 shows the PR curve for both the approaches. The multi encoder deep learning approach proposed in the paper was able to reach PR curve AUC of 26 that is 62.5 % more as compared to logistic regression AUC.

4.1.2 Review Ranking capability of code review model

To assess review ranking ability of our model, we generate only 49 -ve pairs for each positive pair $\langle C_i, R_i \rangle$ in val_dat using same technique explained in section 3.2.2. We then use DeepMEM and baseline method to compute $S_{relevance}$ score and use this score to rank all 50 pairs. Using

this ranking, we calculated two metrics: 1) Mean reciprocal rank; 2) Recall@k.

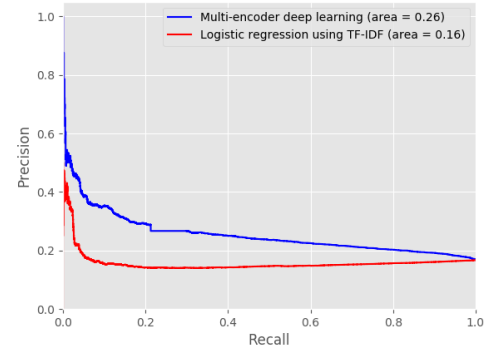


Figure 4 Precision recall curves for Multi-encoder deep learning model (DeepMEM); and baseline method using TF-IDF and logistic regression. The area under the curve for DeepMEM is 26 and for baseline method is 16.

Mean reciprocal rank measures the ranking quality by assuming that there exists only one relevant review, hence it penalizes all other reviews apart from R_i . We calculate MRR (Mean reciprocal rank) as

$$MRR(R_i) = \frac{1}{n_{test}} \sum_{i=0}^{n_{test}} \frac{1}{rank(R_i)}$$

where n_{test} is number of positive pairs in test_dat, $rank(.)$ function computes the index of R_i in ranked list of 50 code review pairs. MRR close to 1.0 is good which indicates that for each pair models ranked right review at first position.

MRR is calculated for both the methods multi encoder, and the baseline method. Figure 5 shows the MRR results. The multi-encoder deep learning achieved 136% higher MRR than logistic regression using TF-IDF. Our model was able to achieve good MRR because of its capability to learn complex code patterns by analyzing sequence of code and review tokens- the capability not present in TF-IDF representation.

MRR works on the strict assumption that there exists only one relevant review. But our data can have more than one review that are of same type and applicable to C_i . Hence, during random sampling process, some -ve pairs might end up having review of same type as R_i . To take into account that there may exist multiple relevant reviews in the dataset, we calculated another metric called recall@k. This metric calculated fraction of $\langle C_i, R_i \rangle$ in val_dat with $rank(R_i) \leq k$ where $rank(R_i)$ is calculated in a same way as explained above for MRR calculation. Figure 6 shows the Recall@k for different k where $k \leq 49$. The proposed model out-performed logistic regression model by ~200% for $k=0,1,2,3,4,5$. Table 3 shows details on individual values of recall@k.

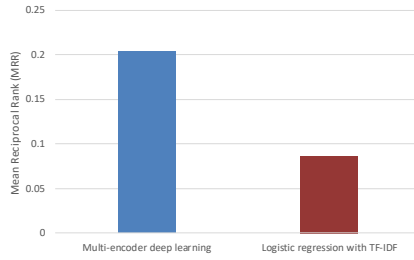


Figure 5 Mean reciprocal rank (MRR) for DeepMEM, and baseline method using TF-IDF with logistic regression

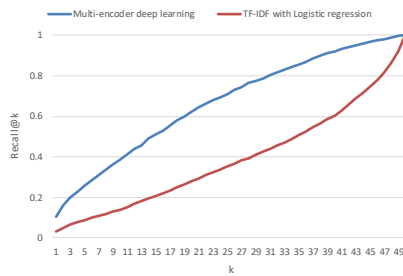


Figure 6 Recall@k for DeepMEM, and baseline models

Table 3 Comparison of Recall@k for DeepMEM and baseline methods.

k	Recall@k for DeepMEM	Recall@k for logistic regression with TF-IDF	%gain compared to baseline
0	0.106	0.031	242%
1	0.159	0.049	224%
2	0.197	0.066	198%
3	0.227	0.078	190%

4.2 User study

4.2.1 Setup

To assess the usefulness of the DCR system, we perform user study with a group of 9 Software developers from within our organization. Distribution of developers according to their experience is shown in Table 4. For the study, we used DCR system to recommend reviews for all the lines in the modified code file in a PR. To generate recommendations for a file, we first split the file into smaller code snippets. For a given code snippet, we followed the process explained in section 3.3 to generate the review recommendation.

We developed a chrome extension that renders the recommendations provided by our system on the specific commit of a PR.

Table 4 Shows distribution of users according to experience level.

Experience level	# of users
Junior Engineers	2
Senior Engineers	5
Principal Engineers	1
Manager	1

4.2.2 Task

Each user was given a set of random commits for files in val_dat and was asked to use the tool to analyze DCR recommendations and select all the relevant recommendations. During the task, our chrome extension collects click data such as click to see recommendation, click to select relevant recommendations, and click to submit selected relevant recommendations. At the end of the study, each user was asked to fill a survey about our system.

4.2.3 Analysis of data collected during user study

During the user study users evaluated around 40 types of reviews recommended by our system and clicked on 216 recommendations out of which they found 117 as relevant recommendation, and 99 as not relevant recommendations. **This resulted in an overall recommendation acceptance rate of 54% across all the 9 users.**

4.2.4 Survey Results

We asked following questions to the users in our survey.

Q1. Is the tool useful?

This question asked users to rank tools usefulness. 7 users out of 9 found the tool useful, and 2 users were neutral. None of the users found this tool not useful.

Q2. Would you recommend this system to colleague?

6 out of 9 said yes to this question. This indicates that users had high affinity towards promoting our system.

Q3. What did you like about the tool?

This question was asked to see if the user responses are correlated to the systems use case to help both the developers and reviewer to perform focused code reviews. As the ones who created the PR, they liked that they can now correct possible common review comments proactively without waiting for a human reviewer's comment. Also, they like the issues found by the system that a static analyzer cannot find; Some examples of such issues include suggestion to implement idisposable interface, add try-catch, and to catch specific exception.

Also, when they switched roles from developer to a code reviewer, they found this tool helpful in performing focused code reviews where they can review code for functional issues without worrying about common code issues.

Q4. What did you not like about the tool?

The users indicated that they would like to see these recommendations earlier during the code development.

5 Discussion and conclusion

We have presented a flexible, adaptive, and easy-to-configure automatic code review system using deep learning called DeepCodeReviewer(DCR). It automatically learns code analyses corresponding to the human reviews on code snippets in historical peer reviews. Code and reviews are pre-processed to remove noise and convert them to sequences of tokens. Then a deep neural network (DeepMEM) is used to analyse and classify relevant and non-relevant code and review pairs. When validated on 18,306 code and review pairs, our model beats baseline model by 62.5% in PR AUC (Figure 4). Given a new code to review, DCR uses the developed model to rank a set of candidates reviews from a repository of common reviews, and selects ones that cross a set threshold. The DeepMEM model when validated for its ranking capability surpassed baseline by 136% in MRR(Figure 5), and ~200% in Recall@k (Figure 6). In the user study to evaluate over- all system, DCR's recommendations had acceptance rate of 54%.

During pre-processing we filtered noise to get a dataset of actionable reviews only. Reviews such as questions, praise related, and complex context related are removed. We also removed longer reviews as they are usually specific and do not occur frequently in training dataset. Using smaller reviews helped our model to learn code patterns better and understand full review text well without any issues related to forgetting beginning parts of it. Our approach converts each code and review token to a dense feature representation using word2vec. This equips our model with semantic information, and avoids sparsity issues related to shorter reviews.

Whenever a developer submits a new code change in a pull request, our system can provide code reviews automatically for common issues. This helps developer to proactively correct issues and helps reviewers to perform focused code reviews towards finding defects. We plan to make DCR learn continuously from the new pull requests, and add information related to code semantics and long-range context. This will make DCR to learn new issues, personalize itself to a team/repository, and learn complex issues.

REFERENCES

- [1] Ackerman, A. F., Fowler, P. J. and Ebenau, R. G. Software inspections and the industrial production of software. In *Proceedings of the Proc. of a symposium on Software validation: inspection-testing-verification-alternatives* (1984). Elsevier North-Holland, Inc., Darmstadt, Germany.
- [2] Ackerman, A. F., Buchwald, L. S. and Lewski, F. H. Software inspections: an effective verification process. *IEEE software*, 6, 3 (1989), 31-36.
- [3] Bielik, P., Raychev, V. and Vechev, M. *Learning a Static Analyzer from Data*. Springer International Publishing, City, 2017.
- [4] Sadowski, C., Gogh, J. v., Jaspan, C., S. E., #246, derberg and Winter, C. Tricorder: building a program analysis ecosystem. In *Proceedings of the Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015). IEEE Press, Florence, Italy.
- [5] Mäntylä, M. V. and Lassenius, C. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35, 3 (2009), 430-448.
- [6] Bacchelli, A. and Bird, C. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the Proceedings of the 2013 International Conference on Software Engineering* (2013). IEEE Press, San Francisco, CA, USA.
- [7] Gers, F. A., Schmidhuber, J. and Cummins, F. *Learning to forget: continual prediction with LSTM*. Institution of Engineering and Technology, City, 1999.
- [8] Boehm, B. and Basili, V. R. Software defect reduction top 10 list. *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*, 34, 1 (2007), 75.
- [9] Fagan, M. *Design and code inspections to reduce errors in program development*. Springer, City, 2002.
- [10] Kennedy, N. *How Google does web-based code reviews with Mondrian*. City, 2006.
- [11] Tsotsis, A. *Meet phabricator, the witty code review tool built inside facebook*. City, 2006.
- [12] Gomes, I., Morgado, P., Gomes, T. and Moreira, R. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal* (2009).
- [13] Balachandran, V. *Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation*. IEEE, City, 2013.
- [14] Allamanis, M., Barr, E. T., Devanbu, P. and Sutton, C. A Survey of Machine Learning for Big Code and Naturalness. *arXiv preprint arXiv:1709.06182* (2017).
- [15] Allamanis, M., Tarlow, D., Gordon, A. and Wei, Y. *Bimodal modelling of source code and natural language*. City, 2015.
- [16] Bhatia, S. and Singh, R. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129* (2016).
- [17] Campbell, J. C., Hindle, A. and Amaral, J. N. *Syntax errors just aren't natural: improving error reporting with language models*. ACM, City, 2014.
- [18] Wang, S., Chollak, D., Movshovitz-Attias, D. and Tan, L. *Bugram: bug detection with n-gram language models*. ACM, City, 2016.
- [19] Sutskever, I., Martens, J. and Hinton, G. E. *Generating text with recurrent neural networks*. City, 2011.
- [20] Karpathy, A. and Fei-Fei, L. *Deep visual-semantic alignments for generating image descriptions*. City, 2015.
- [21] Iyer, S., Konstantas, I., Cheung, A. and Zettlemoyer, L. *Summarizing source code using a neural attention model*. City, 2016.
- [22] Rabinovich, M., Stern, M. and Klein, D. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).
- [23] Allamanis, M., Brockschmidt, M. and Khademi, M. Learning to Represent Programs with Graphs. *arXiv preprint arXiv:1711.00740* (2017).
- [24] Bird, S. and Loper, E. *NLTK: the natural language toolkit*. Association for Computational Linguistics, City, 2004.
- [25] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. and Dean, J. *Distributed representations of words and phrases and their compositionality*. City, 2013.
- [26] Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [27] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15, 1 (2014), 1929-1958.
- [28] Shang, J., Liu, J., Jiang, M., Ren, X., Voss, C. R. and Han, J. Automated phrase mining from massive text corpora. *arXiv preprint arXiv:1702.04457* (2017).
- [29] Chollet, F. *Keras*. City, 2015.