Spotlight: A Machine Learning Approach to Facilitate Code Review

Shruti Sharan
shruti5596g.ucla.edu
University of California, Los Angeles

Haiqi Xiao
haiqi.xiao@ucla.edu
University of California, Los Angeles

Amrutha Srinivasan
amruthas@ucla.edu
University of California, Los Angeles

Hanwen Zhang
hwz@ucla.edu
University of California, Los Angeles

Yuqing Wang
yuqingw96@ucla.edu
University of California, Los Angeles

Aditya V Joglekar
adivj123@gmail.com
University of California, Los Angeles

ABSTRACT

Code review is a form of 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 in the Software Engineering and Open Source Software(OSS) community as a methodology for detection of bugs and other bad practices such as violations of coding standards.

During a modern code review, a reviewer generally writes her/his comments on lines of code that are problematic and need to be changed. In this work, we build models that can understand source code and predict whether a particular line has a high probability of receiving a review comment. We believe that this is one of the first few approaches that uses Natural Language Processing (NLP) tools to understand source code. Our research question is framed as "Using natural language processing, can we predict if a line of code is likely to receive a review comment?"

We evaluate three different models, with their inputs varying in terms of the amount of structural information they contain, to determine if structure and semantics play a major role in the prediction problem. Our findings could be a precursor to more extensive source code analysis using NLP.

1 INTRODUCTION

Code review is a common quality assurance measure in the software industry, where a small team of developers manually read and comment on new code before it is integrated to the code base. It serves as a method to ensure code quality and ensures early defect detection [1],[9].

A lot of research has been done to optimize this process and make it more efficient. Popular tools like Phabricator, Gerrit, Collaborator, and Codeflow have been developed to facilitate code review and make it a more straightforward process. Despite the development of these modern tools, the process of code review takes a lot of manual effort. There are static analysis tools and linters that automate parts of the process. Nonetheless, they only work for detecting some variations of defects and are not very flexible [6]. For a reviewer, no existing tools provides hint to scrutinize certain lines of code, or suggest lines of code that are likely to be commented on, so they spend the same amount of time reading through all changed lines as they would without the help of code review tools. A large proportion of a developer's time is dedicated to reviewing code written by colleagues, and code authors often need to wait a few hours for their code to be reviewed. To make the process more efficient, it is necessary to reduce the time that a reviewer spends on reading the code, and one approach is to mark lines of code that need reviewer's attention, so the potential issues can be found and addressed more quickly. To find the lines of code that need reviewer's attention becomes the goal of our project. From the data set we reviewed, we are able to identify certain patterns in lines of code that are commented by reviewers, so it can be a classification problem for machine learning algorithms. In our study, we assume that there are certain implicit patterns or signatures in the code which make it important for a reviewer that is given the code has a certain makeup, it is statistically more probable that a reviewer is interested in it. If this suggestion is right then this insight could be useful to pick out code lines which might be important for a review. We decided to go with this simple design for now since there has been no pre-existing research in this line and hence we wanted to examine if our idea even makes sense or not. Later, as we will discuss in the the results section - the fact that our simple model delivers a relatively high test accuracy suggests that code indeed seems to contain implicit characteristics (perhaps some kind of syntax, or mistakes,) which makes it more interesting or significant. As with many ML suggestions which often have an intuitive feel- we can say this is what a reviewer does when they are trying to scan the code. The reviewer's training causes them to implicitly scan the code for important code lines, or any patterns deviating from the standard, potential mistakes. To summarize the power of our model is that without explicitly stating them, the model is probably picking features which human reviewers often look for thereby indicating its usefulness towards automating the review pipeline to a certain extent to aid human reviewers do a better job.

We choose Chromium code review data as the data set, which has more than 1 million lines of changed code. In the models we built, we label lines of change that are not commented as negative and lines of change that are commented as positive. Using different models, we try to test the accuracy of our classifier. A classifier with high accuracy is able to tell the code reviewer that an LOC needs more attention, because it is likely to receive a comment. With important LOCs marked, reviewers can save time scrutinizing unimportant changes.

2 RELATED WORK

Code review data analysis using Machine Learning or Natural Language Processing seems to be a relatively nascent research area, and there is not much prior work associated with it. Li et al.[13] describe a study on the applications of deep learning and other

machine learning techniques in the broad field of Software Engineering. Modern code review tools are more effective for facilitating knowledge transfer and collaborative solution to a problem, despite its motivation to find and eliminate defects [4]. Gerrit, a popular open source interactive code review tool, scans initial revision with verification bots which checks for code style with linters, common mistakes with static analysis, and functional issues with tests; code then goes to manual review with the feedback by the bots [15]. Semantic analysis is usually done with a compiler, and not within the scope of code review process. Program analysis using machine learning has become popular more recently. Allamanis, et al. present an extensive survey on of machine learning algorithms used for applications such as code search [2], syntax error correction [5], and detection [7], and bug detection [14]. Our inspiration for this project comes from the work by Gupta et al.[12] in which they present an automatic, flexible, and adaptive code analysis system that recommends code reviews related to common issues in the code based on historical data. Our study basically tries to classify each line in a diff file based on a priority. We define priority here as the probability that a line will be commented on during a code review process.

3 APPROACH

We present Spotlight, a tool to facilitate code review process using machine learning models. It contains complete pipeline to identify the different parts of our problem statement and address them individually. The detailed pipeline of our approach is explicitly shown in Fig. 1. First the chromium web page is crawled to extract code with changed lines. All the changed lines that have a review are extracted and treated as the positive files, since the fact that they have been commented upon means that they have higher importance. The other changed lines of code without any reviewer's comment is treated as the negative class, as they are not as important. After extracting the code at a line level granularity, the dataset is created with the respective class labels of 1 and 0. Once the dataset is ready, the next phase is that of tokenization and embeddings. We tackle three different approaches to understand structural dependence of code with respect to their importance score. In the first approach, we treat the code as raw text. In the second approach we pass it through a Lexer to tokenize the values and use the tokens as input. In the final approach, we add the type of token also in the form of one hot encoding to the actual token, concatenate it and use as input to our Machine Learning model. The model used is a simple model with Bi-directional LSTMs which are used for sequence analysis. Finally the output of our predictive model is to predict if a line of code has a high importance score or not.

3.1 Data Set Exploration

Our classifier is expected work on the level of line of change (LOC). The ideal training data should be in the form of <LOC, label> tuples. Among numerous available code review data sets, CROPS [8] and Chromium conversations[1] are the most extensive. The CROPS data set does not have detailed line-level comment and review. The Chromium data set is the closest to our requirements, because (1) its sheer size - more than 1m lines - allows us to train any model with flexibility, (2) it has line-level comments in the diff file. To get the data we need, we build a crawler to read through the Chromium code review website [2], and extract data for local changes in the format of diff for each changed file, identified by review id, patch id and patchset id.

3.2 Data Extraction

The crawler we developed extracts positive and negative examples separately. Here, by "positive example", we mean a single line of code change on which a review comment has been written, and by "negative example", we mean a single line of code change that has not received a review comment and hence is lower in priority. An example of a diff file we scraped data from is shown in Fig 2. For this project, we only consider review comments on lines that have been added, not deleted. Therefore, our positive examples only consist of added lines for which a comment has been written, and our negative examples are added lines which have not received a comment. Once each line is scraped, we need to apply some additional formatting such as removing diff-specific special characters.

We expect that the number of negative examples to outnumber that of positive examples by a large margin. Because there are much fewer lines that receive a comment than that do not. Hence, the data set is skewed, in the sense that the number of positive examples, in which we are more interested is smaller than that of negative examples. To solve this issue, we have two options :

- Train on the unbalanced data set
- Sub-sample the data so there is an equal distribution of classes

In our approach, we decided to sub-sample the data since even after sub-sampling, the dataset is still large enough for the model to learn from, and also the accuracy would be much more reliable from such a model than one that has been trained on a skewed dataset.

3.3 Data Pre-processing

The field of Natural Language Processing has definitely made huge leaps of progress in the recent years in terms of understanding, interpreting and manipulating human language in its various forms. However, the existing model are not trained to understand source code in the same way that they do languages. In source code, for example, each special character, including spaces have meaning but in natural language processing of human languages, these are simply ignored. The goal of data-preprocessing is to transform data samples into a form which could be used in the machine learning training models. Specifically, we first tokenize codes into a syntactically valid tokens and then retrieve word embeddings for each token using a pretrained model of fastText.

3.3.1 *Code Tokenization*.

We tokenize codes using a tool called pygment lexer [3]. Because our data set only includes C++ files, we employ a CppLexer which is a specific tokenizer in the pygment lexer designed especially to split C++ code statements into lexical units called tokens. For example, given the code snippet below, the result of tokenization of CppLexer in the pygment lexer is shown in Box 1.

[1] https://github.com/meyersbs/chromium-conversations

[2] https://https://codereview.chromium.org

[3] http://pygments.org/

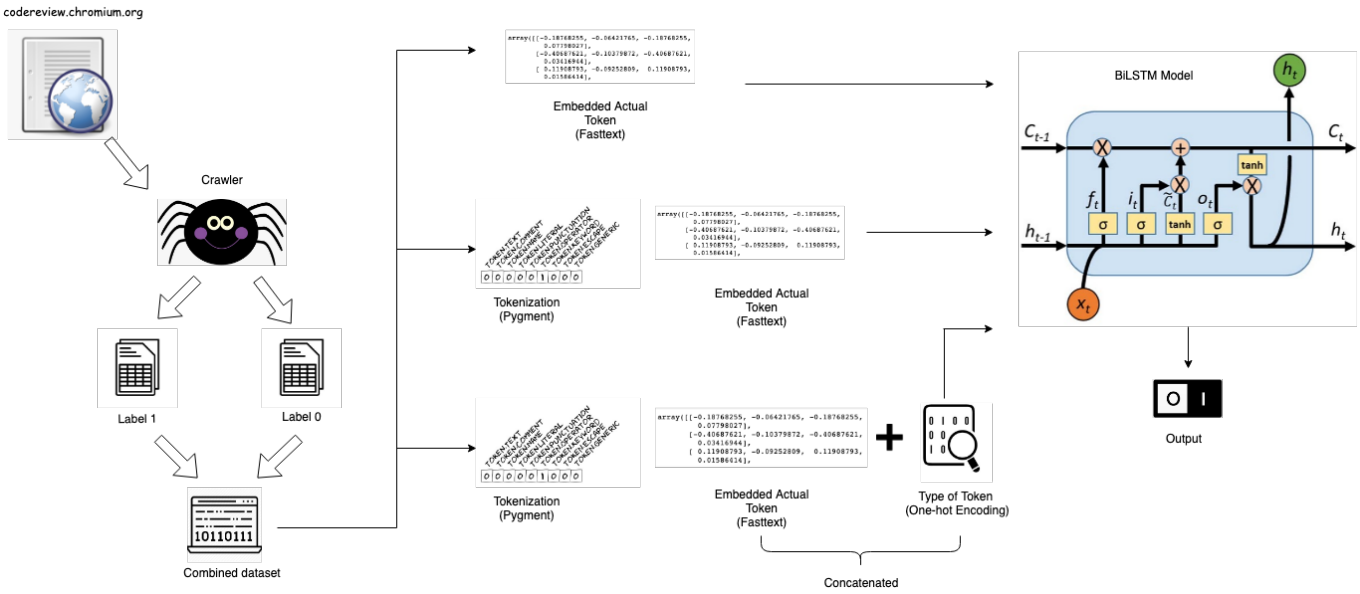

Figure 1: Pipeline of our Approach

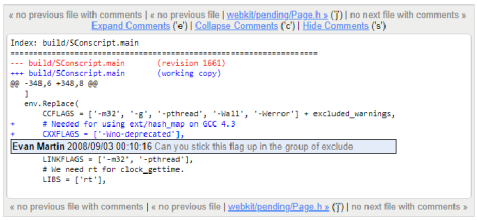

Figure 2: Sample Diff file

```
if(a>0){return a;}
```

```
(Token.Keyword, 'if') (Token.Punctuation, '(') (Token.Name, 'a')
(Token.Operator, '>')(Token.Literal.Number.Integer, '0')
(Token.Punctuation, ')')(Token.Punctuation, '{')
(Token.Keyword, 'return') (Token.Name, 'a')
(Token.Punctuation, ';') (Token.Punctuation, '}')
(Token.Text, '\n')
```

Box 1: Results of Tokenization

As shown in Box 1, CppLexer splits code statement into sequences of tuples where each tuple is made of token name and its values. The token name is a category of the token. According to the tokenization criterion of CppLexer, there are 61 categories of tokens in total which are shown in Table 1. It's worth mentioning that some coding styles such as whitespace and tab will also be preserved when tokenizing.

3.3.2 *Word Embedding*.

After tokenization, sequences of tuples <token_name, token_value> are produced for each line in the training dataset, where token_name belongs to a predefined category of token and token_value is the concrete value. In order to feed tuple sequences into the Deep Learning Model, we use vectors to represent them.

Specifically, for token_name, we use one hot encoding representation because token_name is a type of categorical data. Based on the tokenisation criterion of pygment lexer, there are 61 different token types in total; therefore a vector with the length of 61 is used to represent the token_name. The specific form of vector is $[0, \ldots 1, \ldots, 0]$ where the i-th position is 1 if and only if token_name is the i-th token in the category.

Table 1: Token name categories produced by pygment lexer for C++

Token Names	Some Specific Types
Text	Text
Comment	Comment, Comment.Single, Comment.Multiline, ...
Operator	Operator.Word
Keyword	Keyword, Keyword.Constant, Keyword.Declaration, Keyword.Type, Keyword.Namespace, ...
Name	Name, Name.Attribute, Name.Class, Name.Decorator, Name.Function, Name.Variable, Name.Exception, ...
Literal	Literal.String, Literal.Number, ...
Punctuation	Punctuation
Error	Error

As for token_value, we employ the pretrained word vectors[4] which are trained on Common Crawl and Wikipedia using fastText[5] to retrieve the word embedding vector for each token_value. FastText has an advantage that each word is represented as a bag of character n-grams in addition to the word itself, therefore it can compute the word embedding vector even for a word which is a random combination of characters. For code snippets, it is a big merit because a lot of symbols in the code do not have meanings in natural language. For example, find_mid() which is the name of a function finding the middle value of a list. We cannot just skip these symbols during the pre-processing phase because many of them play an important role in forming syntactically valid and semantically meaningful codes. Hence fastText provides us a feasible way to represent them as vectors, whereas other traditional models like word2vec and Glove fail to do it. In addition to symbols mentioned above, there are some other special symbols such as whitespace, \t and \n that even fastText fails to compute their embedding vectors and directly outputs a vector of all zeroes. It is undesirable because many of them convey some information about coding styles and representing them as zeroes will get the information lost. Instead, we play some tricks to replace them with other characters that have embedding vectors but almost never appear in C++. Specifically, we identify some special symbols that are widely used in C++ and manually construct a small dictionary that maps from these symbols to some other characters that almost don't appear in C++. Then every time we encounter them, we will use the embedding vector of its mapped symbol to represent it. The exact mapping is demonstrated in Table 2.

Table 2: Mapping for Special Symbols

Original Symbols	Mapped Symbols
whitespace	$
\t, \n, \a, \b, \f, \r, \v	@

After getting embedding vectors for both token_name and token_value, we concatenate them together with word embedding vector followed by one hot encoding vector to combine two pieces of information together, and then produce a giant vector with the length of 361 for each tuple in the sequence.

After tokenizing code and retrieving word embedding, the tokenized data is finally in the form of a list of two dimensional vectors. The first dimension specifies the number of tokens in each line, while the second dimension comprises of the respective embeddings of size 361. Since the lines were of unequal sizes we found the line with the largest number of tokens and padded all the vectors to that size. This was done to avoid losing any information and at the same time maintain uniformity of the size. This made all the inputs uniform, with the dimensions (104,361). This sequence of vectors will be further fed into the learning model as input as the next stage in the pipeline.

[4]https://fasttext.cc/docs/en/crawl-vectors.html
[5]https://fasttext.cc/

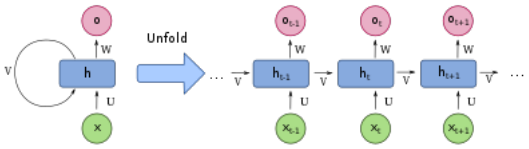

Figure 3: Recurrent Neural Network

3.4 Deep Learning Models

In the domain of Natural language processing, neural networks have found large acceptance due to the mere sizes of these models and their ability to learn inherent underlying information that is difficult for classical algorithms to outperform.

This [3] is very useful in text classification problems since the sequence and order of words in a document, gives a lot of information about its content. For example if the words 'bark' and 'loudly' appear next to each other, it is likely that 'bark' refers to the sound made by a dog. Whereas if the words 'bark' and 'leaves' appear near each other, it is likely, that a the bark of the tree is being talked about [11].

Basic RNNs are a network of neuron-like nodes organized into successive "layers", each node in a given layer is connected with a directed (one-way) connection to every other node in the next successive layer. Each node (neuron) has a time-varying real-valued activation. Each connection (synapse) has a modifiable real-valued weight. Nodes are either input nodes (receiving data from outside the network), output nodes (yielding results), or hidden nodes (that modify the data en route from input to output).

Recurrent neural networks can be thought of as multiple copies of the same network, that pass information along to each other. At each time step, there is some input to the model. After calculations have been made, the output gets sent forward to the next time step and is combined with other input. This is demonstrated in Figure 3. The output of each hidden state of the RNN is dependent upon the output from the previous hidden state, as well as the input to the current state. Being able to pass information along into different time steps is what gives RNN's the ability to make sense of sequential information.

One problem that arises in RNN's is what is known as the vanishing gradient problem. In neural network models, gradient descent is used to find the global minimum of a cost function that is optimal for the network. Information gets passed through the network from neuron to neuron and the error of the network is calculated, which is then back-propagated through the network to update weights. During propagation, every neuron that contributed to the error of the network must have its weighs updated, and since RNN's are sequential, we have to back propagate through lots of hidden layers of neurons. When small weights are multiplied over and over again, the value decreases very quickly, and this is the vanishing gradient problem. The problem with this is when you have a sequence of tokens, you can lose information about words that are far apart, and only track patterns of words that appear close to each other.

One way to solve this problem is to introduce 'gating' which is what Long Short-Term Memory (LSTM) and Gated Recurrent

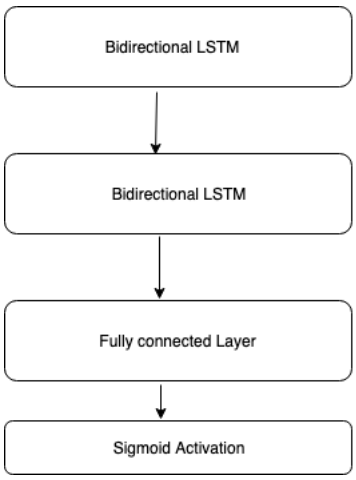

Figure 4: BiLSTM architecture

Units (GRU) do. Gating simply helps the network know when to forget the input, and when to remember it for future time steps. This solves the problem of the vanishing gradient. For our project, we chose the LSTM implementation of gated recurrent networks which were further experimented upon.

3.5 BiLSTM Model

LSTMs are a special kind of RNN, capable of learning long-term dependencies. LSTMs also have the same chain like structure, like the recurrent netwroks, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way. An LSTM has three gates, to protect and control the cell state. The first step in an LSTM is to decide what information is going to be discarded from the cell state. The next step is to decide what new information is going to be stored in the cell state. This is done using two different layers of activation functions. After that the information about the old cell is dropped and the new information is added, as decided in the previous steps.Finally, the output is decided. This output is based on the cell state, but will be a filtered version.

LSTMs are known to perform well for text classification problems but using a bi-directional LSTM tends to do better. This is because context is learnt both ways, in a forward as well as backward manner. Since our dataset involves code, having dependencies preserved from both sides made more sense.

In the experiments performed, the CuDNNLSTM layers from the python package Keras,were used to represent the LSTM layer. Two bidirectional LSTM layers were stacked one after the other specifying the input shape of each sample as (104,361), where 104 were the number of tokens in each sentence and 361 was the size of the embedding vector. A fully connected layer was added after the two bi-LSTM layers with the sigmoid activation function, to get the final output. The model computes a binary cross entropy loss since our aim is binary classification. The adam optimizer which is combination of the adagrad and momentum algorithms, is used to optimize this loss. The model was evaluated on the accuracy metric and run for 5 iterations, each with a step size of 1000. This implies that 1000 samples were subsampled from the entire data at each step. After the model was fit on the training data, cross-validation was performed, to avoid overfitting. After that the test accuracy was computed to see how many test samples were predicted correctly. The detailed architectural design is shown in Figure 4.

4 EVALUATION

4.1 Metrics of evaluation

4.1.1 ***Accuracy***.

Accuracy is one metric for evaluating classification models. Informally, accuracy is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

4.1.2 ***Precision***.

Precision attempts to answer the following question: What proportion of positive identifications was actually correct? Precision is defined as follows:

$$Precision = \frac{True\ Positives}{True\ Positives\ +\ False\ Positives}$$

4.1.3 ***Recall***.

Recall attempts to answer the following question: What proportion of actual positives was identified correctly? Mathematically, recall is defined as follows:

$$Recall = \frac{True\ Positives}{True\ Positives\ +\ False\ Negatives}$$

4.1.4 ***F1 Score***.

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

$$F1 = 2 * (precision * recall)/(precision + recall)$$

4.1.5 ***ROC Curve***.

A receiver operating characteristic curve, or ROC curve, is a plot that illustrates the ability of a binary classifier to diagnose and classify as its discrimination threshold is varied.The ROC curve is created by plotting the true positive rate against the false positive rate at various threshold settings. Essentially, an ROC curve tells us how capable a model is in distinguishing between two classes. AUC (Area under a curve) is another metric which is simply calculated by finding the area under the ROC. Higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s.

4.2 Existing Case Studies

An ablation and Configuration Study has been made to examine the importance and effect of each major component of BiLSTM model. The Stanford Natural Language Inference (SNLI), which contains 570K human annotated sentence pairs, is used as the development and training set. The impact of BiLSTM dimensionality is then studied based on the performance of SNLI. All the settings on the development set is investigated and evaluated.

Figure 5 shows the performance of different configurations of the proposed model on the development set of SNLI along with their p-values comparing to the complete BiLSTM. According to the table, we can see that a single modification of a component

Model	Dev Acc[a]	p-value
BiLSTM	**88.69%**	-
BiLSTM - hidden MLP	88.45%	<0.001
BiLSTM - average pooling	88.50%	<0.001
BiLSTM - max pooling	88.39%	<0.001
BiLSTM - elem. prd[b]	88.51%	<0.001
BiLSTM - difference	88.24%	<0.001
BiLSTM - diff[c] & elem. prd	87.96%	<0.001
BiLSTM - inference pooling	88.46%	<0.001
BiLSTM - dep. infer[d]	88.43%	<0.001
BiLSTM - dep. enc[e]	88.26%	<0.001
BiLSTM - dep. enc & infer	88.20%	<0.001

[a]Dev Acc, Development Accuracy.
[b]elem. prd, element-wise product.
[c]diff, difference.
[d]dep. infer, dependent reading inference.
[e]dep. enc, dependent reading encoding.

Figure 5: Ablation study results

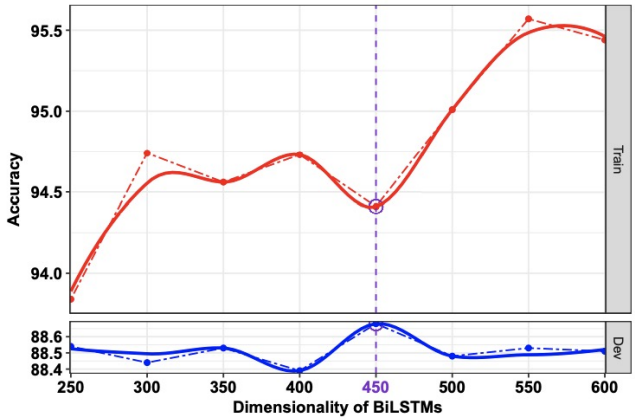

Figure 6: Impact of BiLSTM dimensionality in the model on the SNLI training set (red lines) and development set (blue lines)

leads to a different model that is statistically significant with the p-value smaller than 0.001. According to the statistics, we can see that the removal of any components is harmful for the accuracy of the developments. This proves the effectiveness of these components. Among these components, the max pooling, dependent reading and difference in the attention stage have the stronger contribution to the good performance. Furthermore, the last four study cases prove the fact that the dependent reading in the encoding stage is important due to its ability to focus on more relevant aspects of the sentences.

Figure 6 shows that behavior of the proposed model accuracy on the SNLI dataset. Through the graph we can see that, the BiLSTM model achieves its best performance with 450-dimensional for both the development set and training set. Besides that, the changing phase of red line also shows that the model doesn't work so well when the dimensional value is too low, yet a very high dimensionality like 600 also causes a drop down on the accuracy. Thus, it is suggested to use 450-dimensional BiLSTM model.

4.3 Our Model Evaluation

As is shown, Bi-LSTMs outperform other classical models in terms of text classification. Since we are dealing with a similar problem, Bi-LSTM was the model of choice. Adding two layers of Bi-LSTM followed by a dense layer showed the best results among the simple models we experimented upon. As can be seen, there is a steady decrease in both the the training as well as validation loss. This implies that our model is able to learn properly and optimize the loss in every iteration. The accuracies also show a steady increase. It starts at close to 50% which is intuitive, as it does as badly as a random guess. But over time, as the model trains and learns from the different samples, the accuracy increases considerably, reaching a high of 90% on the training data. When tested on the test dataset, it does decently too. As expected, the baseline model with absolutely no pre processing done and treated as text, does the worst with a test accuracy of 55%. The intermediate model which takes tokens from the CppLexer (Pygment) as the input does much better than the baseline. This is expected, as the tokenizing adds structural constructs to the learning process. Finally it is observed that the third model, with the types of tokens encodded along with the actual tokens, performs the best. A conclusion could then be made that the added information of what type of token it is (Eg operator, keyword, symbols) makes the learning process even more holistic. The Roc curves also show improvement across the models and the F1 scores are high, implying that the model is able to distinguish the two kinds of code, ie, with and without reviews. In general, the results are in accordance with our initial hypothesis.

5 VALIDITY ANALYSIS

5.1 Evidence of Construction Validity

One of the major threats of our design resides on the construction validity. Our research is based on a simple cause and effect relationship. The relationship assumes that, if there is a problematic line of code exists on program A (whether we take account of the context or not), it is likely that a similar line of code in program B, is problematic. It makes sense in some degree: there are common characteristics on erroneous lines as programmers are likely to make similar mistakes during the process of development. [6] For example, a line " if(x + y < 0) " seems to be a erroneous line if we are sure that x and y represents "number of people," which must be positive, yet the line with similar syntax (if statement) and the similar context (number of people) " if(w + z < 0) " should also be problematic.

5.2 Threats of Validity

There are several threats to validity in our studies. An external threat exists that we only performed all our studies on one code project. However, this project is a real one, and is still being actively worked on by programmers worldwide. The review comments and corresponding diff files are from the actual code review process of the Chromium project. Threats to internal validity mostly concern the fact that we base our analysis on the assumption that the model predicts if a line of code has a higher chance of being commented on the basis of its structure. There could be other factors that the

[6]https://www.softwaretestinghelp.com/types-of-software-errors/

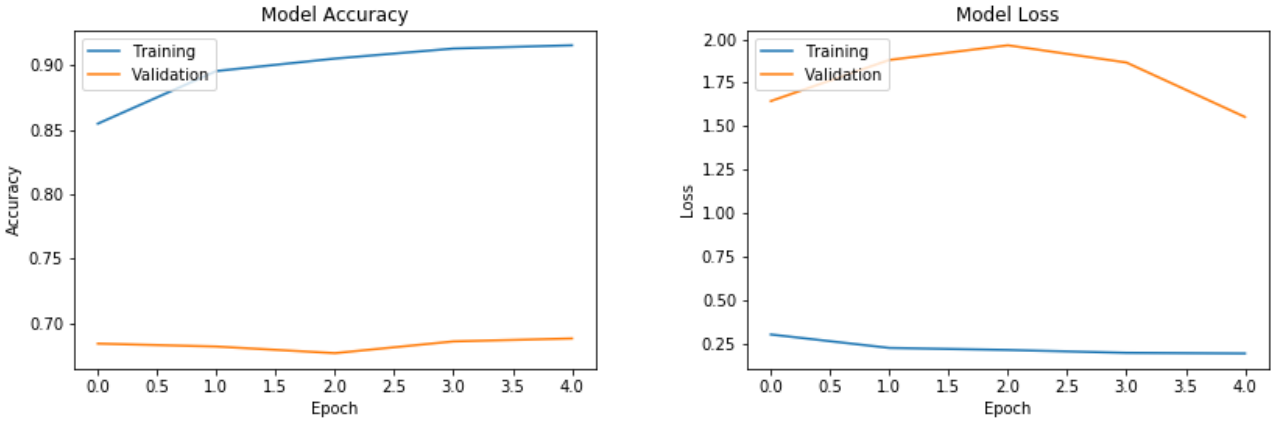

Figure 7: Accuracy and Loss of Model on Training and Validation Dataset

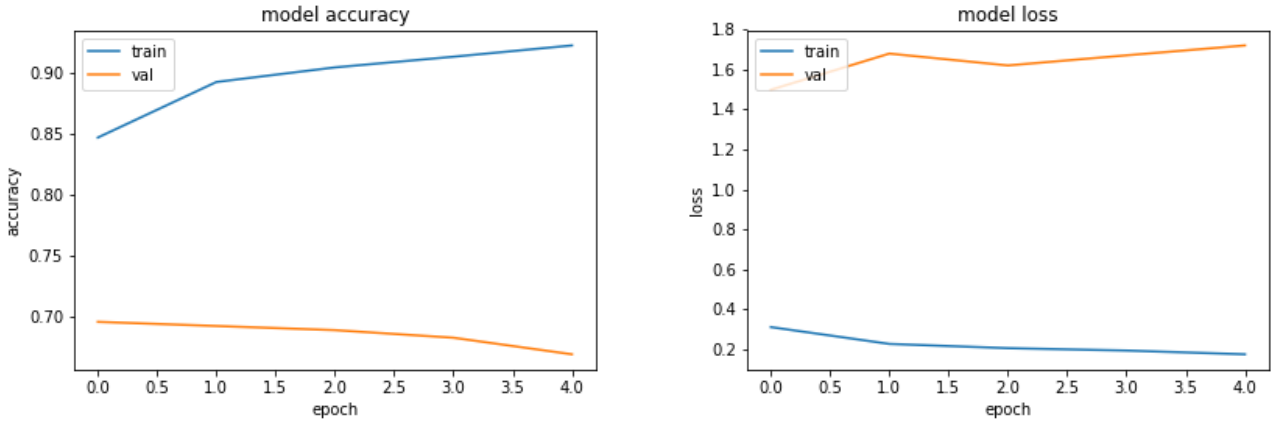

Figure 8: Accuracy and Loss of Model without One Hot Encoding on Training and Validation Dataset

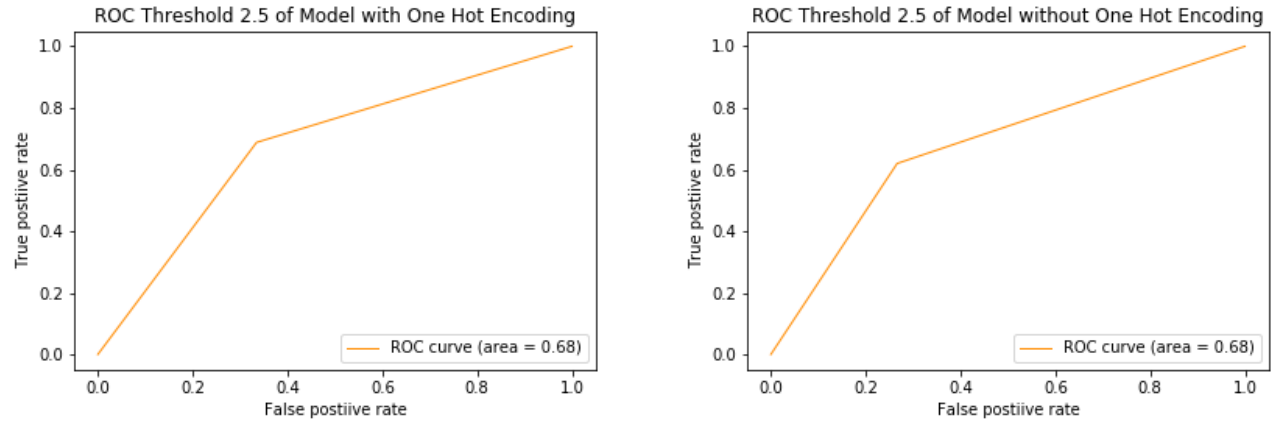

Figure 9: The ROC of our model in comparison to the baseline model

Table 3: Accuracy and Loss of Model with and without One Hot Encoding on Training, Validation and Testing Dataset

	With One Hot Encoding		Without One Hot Encoding	
	Accuracy	Loss	Accuracy	Loss
Training	91.54%	0.1913	92.30%	0.1762
Validation	68.84%	1.5531	66.91%	1.7199
Testing	67.68%	1.6041	67.58%	1.6920

Table 4: Performance of Baseline using fastText

	Accuracy	**Precision**	**Recall**
fastText	55.50%	55.50%	55.50%

	F1 Score
Model without One Hot Encoding	0.66
Model with One Hot Encoding	0.6837

model may be learning to predict this. However, we can observe a steady increase in accuracy among the three models that were experimented upon. The first was one where the input was passed in as raw text, but each of the other two add an additional structural component to the input, and correspondingly have a higher only.

6 CONCLUSION

The code review tools that are widely used have their limitations. For example, most of them evaluate in terms of line-level differences only, instead of focusing on a more holistic approach. Moreover, these code review tools are also hard to understand due to the large patches, cluttered changes across different files in the code [10]. Surveys also shows that, some programmers prefer the primitive methods like printing statements or reading the code, than using code review tools because they don't see the value from it. Taking these facts as our motivation, we've decided to develop Spotlight, an automated code review tool that detects lines with potential problems using machine learning.

Spotlight uses Chromium web review data as training and testing data set. It includes a web crawler that is able to extract the positive and negative local changes of line. We used the pygment lexer [12] to tokenize the code, transferring it to the vector array that machine could understand. Then these tuples are processed by the model we designed, which is pre-trained on Common Crawl and Wikipedia using fastText. After obtaining the word embedding, these vectors are then put into deep learning models. In the end, a separate testing data set was used to test the accuracy of the deep learning model.

As the result has shown, Spotlight reaches a testing accuracy of 67.7%. This means that the algorithm does much better than a random guess. If the algorithm took a random guess each time, the accuracy would be 67.7% on an average. Plus, we have a steady decrease in loss, resulting at a final value of 1.60, which implies that the neural network model is able to learn code structures after each iteration of optimization.

To address our research questions, we can see that each model has a higher accuracy with added structural information. This validates our hypothesis that code structure does have a significant correlation with the probability of a changed code line being reviewed.

7 FUTURE WORK

A logical addition to this study would be to train on the entire data set. The data set in its entirety is extremely skewed in the sense that the distribution of the classes is is not equal. The negative samples far outnumber the positive samples. The entire imbalanced data set would be a much more realistic representation of what code review data will look like in a real world scenario, since there will always be more negative than positive samples. It is also important to evaluate the resultant models on metrics other than accuracy since accuracy will be deceptively high owing to the skew in the data set. Our next step would be to try an AST based approach where instead of the input being code tokens, it could be the AST of the code being reviewed. Our bigger picture for this project is to ultimately produce models that can intelligently detect not only the location of a possible review comment, but also the type of comment and generate one automatically.

Note that the current model doesn't identify what exactly made a line important, that is it is just a binary classification. Future work could focus on identifying categories of review elements. An even more ambitious goal is to generated a natural language review for each important line detected. This presents two main challenges.

- **Multi-class classification:** Diagnostic multi-class problems in general are much harder in general. Simply because they not only have to identify if there is a problem/anomaly/interesting element present in the input but also figure out the category in question.
  This conceptually is a much harder problem to solve (conceptual complexity). Further multi-class training has issues like class imbalance which may lead to spurious models. But this extension for review automation will be highly useful because it will spare the reviewer trouble of trying to figure out why a particular line has been flagged as significant by the above algorithm.
- **NLP based review-generative models:** These models have rekindled hopes of one day being able to completely automate code review. These require much more sophisticated sequence-to-sequence algorithms, a lot more computational resources and training data. This is because patterns in natural language are much more sparse than categorical/binary variable data we use in this study

REFERENCES

[1] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. 1989. Software inspections: an effective verification process. *IEEE software* 6, 3 (1989), 31–36.

[2] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[3] Peter J Angeline, Gregory M Saunders, and Jordan B Pollack. 1994. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks* 5, 1 (1994), 54–65.

[4] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 712–721. http://dl.acm.org/citation.cfm?id=2486788.2486882

[5] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129* (2016).

[6] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a static analyzer from data. In *International Conference on Computer Aided Verification*. Springer, 233–253.

[7] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 252–261.

[8] Open Source dataset. [n. d.]. Code Review Open Platform. https://crop-repo.github.io/#structure

[9] Alastair Dunsmore, Marc Roper, and Murray Wood. 2003. Practical code inspection techniques for object-oriented systems: an experimental comparison. *IEEE software* 20, 4 (2003), 21–29.

[10] Rigby et al. 2008. Open Source Software Peer Review Practices: a Case Study of the Apache Server. *ICSE* (2008).

[11] Samuel Gabbard, Jinrui Yang, and Jingshi Liu. [n. d.]. Quora Insincere Question Classification. ([n. d.]).

[12] Anshul Gupta and Neel Sundaresan. [n. d.]. Intelligent code reviews using deep learning. *KDD18 Deep Learning Day* ([n. d.]).

[13] Xiaochen Li, He Jiang, Zhilei Ren, Ge Li, and Jingxuan Zhang. 2018. Deep Learning in Software Engineering. *CoRR* abs/1805.04825 (2018).

[14] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 708–719.

[15] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2018. An Empirical Study of Design Discussions in Code Review. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18)*. ACM, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3239235.3239525