Deep Learning for Code Repair

Defect Repair Prediction using Deep Learning

Nancy Pang
nancy.pang@alumni.ubc.ca
The University of British Columbia

ABSTRACT

Identifying quality issues in software is a crucial but expensive task for developers to take on. To help developers identify quality issues, Automated Bug Detection (ABD) is widely used in practice. While existing ABD systems are useful, creating a bug detector is a time consuming and highly manual process. Bug detectors must be precisely specified to ensure low rates of false positives and upon identification of a bug, a repair action must be proposed to the developer. To address the high cost of developing static bug detectors, we propose a new deep learning approach to building ABD systems that (1) automatically learns bug detector rules and (2) generates patch suggestions using examples of correct and incorrect code mined from open source JavaScript projects. As a proof of concept, we use our approach to find and suggest repairs for a common JavaScript API misuse bug in the `jQuery.ajax` API. Applying our approach to a corpus of 27,000 instances shows promising results. On an evaluation of real-world bugs, our bug detection model had a low false positive count with 60% precision and 86% recall. Correct repair patches were generated for most bug instances. With a BLEU score of 84.0, the patches were also similar to human-written patches. Our findings can be applied to existing tools to prevent common bugs from being introduced into code bases and used to streamline the development process by providing developers with actionable fixes.

CCS CONCEPTS

• **Software and its engineering** → *Empirical software validation*; *Automated static analysis*;

KEYWORDS

Deep learning, Sequence Learning, Bug patterns, Defects, Javascript, Repair

1 INTRODUCTION

Software bugs are a costly problem that continue to drive up development costs and plague businesses. A recent study conducted by Tricentis, an Australian firm, shows that bugs cost software companies up to 1.1 trillion dollars annually [1].

To reduce costs many companies use Automated Bug Detection (ABD) techniques. The most popular technique used is Static Analysis (SA) which searches source code for known bug pattern instances. Traditionally, SA tools are offered as frameworks, with an extendable set of rules which detect bug pattern instances [2, 3]. SA is able to catch common bug patterns, however, many bug patterns still pass through undetected due to the non trivial and highly manual process of creating bug detectors. In addition, bug detectors need to be manually tuned to decrease the reported number of false positives to provide value to developers.

The rise of richly interactive web applications, as well as the advent of Node.js as a backend runtime, has drawn developers toward using JavaScript making it a popular language for client and server side applications. However, with JavaScript being dynamic and asynchronous, it is both time consuming and difficult to detect quality issues making the language especially vulnerable to bugs.

To address these challenges with existing ABD tools we propose an approach for automatically learning bug detection rules and repairs using JavaScript code examples. Leveraging deep learning techniques we learn structural and semantic information from code. Structural and semantic information are highly useful since knowing the location(structure) and context(semantics) of an error allows for more accurate bug predictions and repair suggestions. A sequence to sequence model approach is used to model the sequential structure of code and an encoder decoder architecture used to learn features that distinguish between correct and incorrect sequences of code.

As a proof of concept, we evaluated or approach for detecting and suggesting repairs for a common API missuse of the `jQuery.ajax` API shown in Listing 1. We mined open source repositories for incorrect and correct usages of the `jQuery.ajax` API. During training, these incorrect inputs and correct outputs were provided to help the model learn the repair, shown in Listing 2. To evaluate our tool, real-world bugs were fed into the model and the outputted repair sequence was manually checked for correctness. Training on 24,706 examples, our model achieved a BLEU score of 84.0, precision of 60% and recall of 86%.

The contributions of this thesis are:

(1) A novel technique for automatically learning bug detection and repair rules from real life examples.
(2) A proof-of-concept implementation targeting a common `jQuery.ajax` API bug pattern (see Listing 1).
(3) An evaluation of the technique on real bugs mined from open source repositories.

2 BACKGROUND AND MOTIVATION

Static Analysis (SA) tools are widely used in industry for discovering defects early in the development process [4]. They find defects by searching through source code for violations of rules that have been pre-defined by Static Analysis authors. Specifying these rules is a highly manual and time consuming process, because static analysis authors must ensure that their tools (1) have a low false positive rate and (2) produce actionable alerts [10, 11].

One of the most challenging aspects of specifying defect detection rules is ensuring a low proportion of false positives. False positives occur when the defects detected by the rule are not perceived as bugs by the user. To limit false positives, bug detection rules must be precisely specified, which often includes context from actual use cases after the static analysis tool starts being used [18].

Another challenge to Static Analysis development is ensuring that developers can understand the alerts generated by the tool and create a fix [10]. One way static analysis tools can solve this problem is by suggesting quick fixes that assist developers by giving them the option to apply a pre-computed patch. However, creating automated patches, especially ones that are similar to ones that developers may create, is difficult. Repairs depend heavily on both the type of bug and its context within the surrounding code.

2.1 JavaScript Static Analysis

Programs written in JavaScript are particularly difficult for static analysis tools because of JavaScript's dynamic nature. Loose typing, asynchrony and unnamed, variadic parameters make JavaScript especially vulnerable to bugs and create unique challenges for static analysis.

Consider the following example of a bug pattern which is unique to JavaScript. jQuery's `$.ajax` API method is frequently used for client and server side communication and documented well. However, many developers still have trouble using this API feature correctly. One common issue developers run into is shown in Listing 1, this is a real bug pattern mined from an open source repository [8]. In this case, a developer wants to post JSON content to a server. The arguments for the call are specified through a JavaScript object, as is common with JavaScript APIs. The developer correctly specifies the `contentType` as `application/json`, however, the developer incorrectly specifies `data` as an object literal. The object literal must first be serialized with a function such as `JSON.Stringify`, as shown in Listing 2.

Listing 1: Ajax Post Bug Pattern

```
  $.ajax({
    url: `/' + $scope.project.name + `/branches/',
    type: `PUT',
    data: { branches: list },
    contentType: `application/json',
    dataType: `json',
    success: function(res, ts, xhr) {
      $scope.success(res.message, true, false)
    },
    error: function(xhr, ts, e) {
      if (xhr && xhr.responseText) {
        var data = $.parseJSON(xhr.responseText)
        $scope.error(`Error adding branch: ' + data.errors[0], true)
      } else {
        $scope.error(`Error adding branch: ' + e, true)
      }
    }
  })
```

Listing 2: Ajax Post Bug Fix

```
  $.ajax({
    url: '/' + $scope.project.name + '/branches/',
    type: 'PUT',
    data: JSON.stringify({ branches: list }),
    contentType: 'application/json',
    dataType: 'json',
    success: function(res, ts, xhr) {
      $scope.success(res.message, true, false)
    },
    error: function(xhr, ts, e) {
      if (xhr && xhr.responseText) {
        var data = $.parseJSON(xhr.responseText)
        $scope.error('Error adding branch: ' + data.errors[0], true)
      } else {
        $scope.error("Error adding branch: " + e, true)
      }
    }
  })
```

Exisiting Static Analysis tools struggle to detect bug patterns that require context understanding, when elements are dependent on or affected by other elements in source code, the relationship between `data` and `contentType` in Listing 2 is an example of this dependency. Existing tools are unable to map dependencies therefore they expect developers to explicitly specify rules (eg. regex of patterns, string sequences) that are allowed and disallowed which are then asserted on a program's AST nodes. Writing a checker for the pattern in Listing 1 is non trivial and time consuming. First listeners must be registered to detect nodes that invoke the `$.ajax` function and the type of operation is verified to be `POST`. From there the `contentType` is then checked, if it is type is `application/json` the `data` field is asserted to begin with `JSON.stringify`. A lot time and manual work was needed to specify this rule; `data: JSON.stringify(...)` given `contentType: application/json` and `$.ajax POST`.

2.2 Automating Rule Specification

Due to the difficulties faced by static analysis authors, it is desirable to remove the manual process of specifying bug detection rules. Our insight is that by using techniques pioneered by natural language translation, we can automatically learn bug detection rules from examples and assist developers by automatically suggesting fixes.

Software code is an artificial language with a set of rules and constraints, it shares many characteristics with text found in natural languages, a code token can be thought of as analogous to a word in text. This allows us to leverage language understanding and machine translation techniques, and apply it to our task of defect prediction and repair suggestion. Some interesting characteristics of code are:

(1) *Repetitive Sequences:* Certain code constructs occur very often. Single control flow tokens like if, else, while and sequences of tokens like `for(var i; i < n; i++)` or `console.log(...)` are commonly found in source code.
(2) *Structural Information:* Code has explicit structural information, these can be complex hierarchies like nested control loops or deep function call stacks. Structure is also implicit in a token sequence, the specific order of tokens matter.
(3) *Context:* Long term dependencies exist in code, a code token may depend on or impact other code tokens which are not immediately preceding or succeeding it. Code constructs that come in dependent pairs such as, try and catch, or if and else, can be spaced arbitrarily far apart from one another in code.

Applying these observations we can treat the bug pattern shown in Listing 1 as a "token paragraph", context can be learned by modelling the relationships between "token sentences", specifically, the relationships between `$.ajax, contentType, data` elements.

3 APPROACH

This section presents the process for automatically creating bug detectors and generating repair suggestions through deep learning. The high level idea is to learn the features that distinguish between common bug patterns and their corresponding fixes.

Our approach extracts bug pattern instances from a corpus of source code, processes the instances into token sequences, and then generates datasets that consist of token sequences to help train the deep learning model. The proposed approach consists of several steps and is illustrated in Figure 3.

(1) *Mine code corpus:* Extract examples of bug patterns and their respective fixes from open source repositories.
(2) *Create vocabulary:* Map code constructs to tokens and build vocabulary from token set.
(3) *Abstract code:* Use vocabulary list to abstract function, identifier and literal names.
(4) *Train Encoder Decoder* Train encoder decoder on datasets to learn bug pattern features.

3.1 Mine Code Corpus

Code examples are mined from commits in open source repositories. A commit is a transformation on the current state of code, a change pair captures the before and after state of the commit. There are three types of change pairs:

(1) *Repair:* Indicates that the before sequence is different than the after sequence. For our purposes the before is considered buggy and the after sequence correct. These are real code examples mined from repositories.
(2) *Mutant Repair:* Similar to repair however mutant repairs are generated repairs where correct code is mutated into a bug. This is done to create more training examples for the model.
(3) *Nominal:* Indicates that there is no change between the before and after sequences. This is used to train model to prevent it from modifying correct code.

We extract pairs by using an AST diff utility on the before and after state of a change pair to extract differences between code examples. Consider the example in Listing 3, where a call is made to a method which expects three arguments. Calling this method with less arguments will result in incorrect behaviour and is considered a bug, an example of which is shown in Listing 4. Converting the code to its AST representation, shown in Figure 1, and finding the difference between the buggy AST in Figure 1a and the correct AST in Figure 1b reveals that the correct AST has an additional argument `c`.

Listing 3: Correct Method Invocation

```
result = method(a, b, c)
```

Listing 4: Incorrect Method Invocation

```
result = method(a, b)
```

3.2 Create Vocabulary

Developers are free to express their creativity when coding, this leads to variability in function names, identifiers, and literals in source code, it also creates an interesting challenge, namely it is impossible to cover the entire scope of developer defined names. A data processing step helps the model figure out which defined names are relevant and important to preserve, it ensures that variables are standardized. The top N names, calculated by usage frequency, are tracked and form the vocabulary list. Commonly used control statements, punctuation and API functions are captured within the vocabulary list while a developer's stylistic statements are not captured. The vocabulary list is provided to the encoder decoder model as a list of recognized tokens.

3.3 Abstract Code

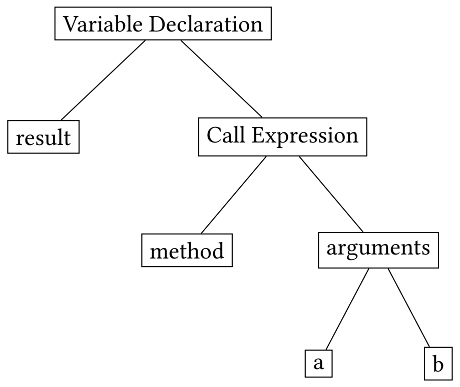
3.3.1 Abstract Functions. To deal with the issue of nested function definitions, which can manifest as extremely long token sequences, they are abstracted. Take for example a nested function call shown in Listing 5, this is abstracted into Listing 6. Functions are replaced with a special abstract token, `function()`. This is done using structural information obtained from the sequence's AST.

Listing 5: Original Nested Function

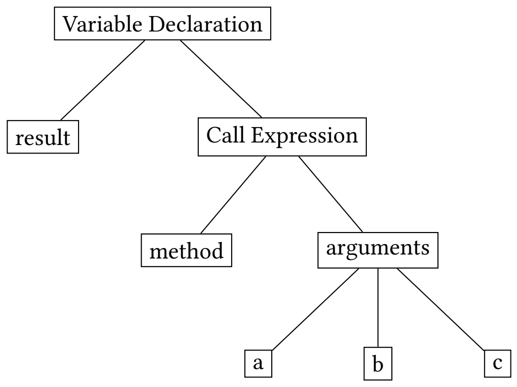
```
result = function1(
  statement1
  statement2
  function2();
);
```

Listing 6: Abstracted Nested Function

```
result = function(function());
```



(a) result = method(a, b)



(b) result = method(a, b, c)

Figure 1: ASTs of a correct and incorrect method invocation, the difference in ASTs reveals the repair, where parameter C needs to be provided.

3.3.2 Abstract Identifiers and Literals. In order to standardize the names of identifiers and literals used across projects each token sequence is processed against the vocabulary list. If a token is in the vocabulary list it is preserved, if not, it is replaced with a custom abstract token, mapping rules are shown in Table 1. The type of a token, extracted from a sequence's AST, is used to determine the abstract token mapping.

Token Mapping	
Type	Custom Token
Identifier	@name
String	@string
Number	@num
Function	@function()

Table 1: Table of Token Mapping Rules

3.4 Recurrent Neural Networks

A RNN is a probabilistic model that captures the context of sequential data. RNNs are successful at modeling long term dependencies because every computation of state depends on the previous value of state. Our approach uses the encoder decoder architecture [5, 22] to translate between buggy and correct code, both the encoder and decoder modules are special forms of RNNs known as Long Short Term Memory (LSTM) Networks. This approach has many key benefits, the model is trained directly on real life examples of code and able to handle variable length input and output code sequences.

3.4.1 LSTM Networks. LSTM Networks are, a special form of RNNs comprised of chained LSTM units. Each LSTM unit passes state information to its successor allowing information to persist and long term dependencies to be learned. Figure 2 shows a typical LSTM unit. Each unit is responsible for determining what state to forget, update and output [19]. The forget gate, f_t, uses the previous state h_{t-1} and current code token x_t to output a number between 0 and 1. A value of 1 indicates that the previous state is preserved completely while a 0 indicates that everything is forgotten. The update operation is performed by combining the input gate i_t and candidate values C_t. Finally the output gate o_t determines what state to pass onto the next unit by looking at the previous state h_{t-1} and current code token x_t.

3.4.2 Encoder. The encoder is a LSTM network responsible for reading in tokens from a code sequence and encoding a vector representation. The vector representation represents the context and meaning of the code sequence.

3.4.3 Decoder. The decoder is a LSTM Network responsible for using encoded state vectors, created by the encoder, to generate the correct repair sequence. It does this by predicting the next token given the current tokens seen.

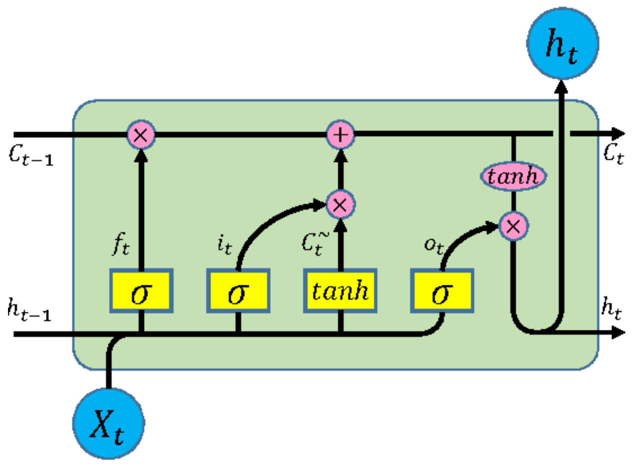


Figure 2: LSTM Unit[6].

3.5 Training Encoder Decoder

Training helps the model learn semantic features that distinguish between buggy and correct sequences of code. To train an effective model many hyper parameters need to be tuned manually. Heuristics from existing work that leverage LSTM networks for NLP and code understanding tasks [7, 12, 15] show that the model is particularly sensitive to these hyper parameters:

(1) Number of Layers
(2) Number of Units
(3) Regularization
(4) Number of Training Steps

During training, source and target sequences are fed to the encoder and decoder. The source is a buggy token sequence and the target is the expected correct token sequence. The encoder generates a state vector (s_e) that captures the semantics of the buggy sequence. The initial state of the decoder is set to (s_e), the decoder then generates the correct token sequence one token at a time.

The decoder generates a token, for a given time step, by selecting the most likely token to appear from the defined vocabulary set. The probability of a token (P_i) is computed at each time step based on the context of past tokens ($token_{1:t-1}$) where $token_{1:t-1}$ = ($token_1$, $token_2$ ... $token_{t-1}$). This is illustrated in Equation 1, V is the vocabulary set and $P_i \in \mathbb{R}^V$. This probability vector is created for every token in the vocabulary.

$$P_i(token_t | token_{1:t-1}) \quad (1)$$

The Cross Entropy Loss shown in Equation 2 is computed on each token in the sequence to determine if the right token was selected for that time step. For individual token loss, a loss of 1 indicates a mismatch, a loss of 0 indicates the right selection. The individual token losses are summed to produce the overall loss for the sequence.

$$-\log \mathbb{P}(y_1, \ldots, y_m) = -\sum_{i=1}^{n} \log P_i[y_i] \quad (2)$$

Backpropagation through time (BPTT) is used to update the weights and parameters of the RNN. The goal of BPTT is to reduce the training loss of the model. The Adaptive Gradient Optimizer (Adam) is used to update the weights of the model.

3.6 Defect Prediction and Repair Suggestion

Once a model is trained it can detect bugs in input sequences and output repair suggestions. Consider the bug in Listing 1, a bug exists since `contentType: application/json` and `data` has not been serialized with `JSON.stringify`. The model uses its learned dependencies between `contentType, data` to identify that this sequence contains a bug. It then suggests a repair by inserting tokens that differentiate the buggy from correct code. For this specific bug pattern, the suggested repair will contain `data: JSON.stringify(<data>)` in its sequence, where `JSON.stringify` has been inserted by the model.

4 EVALUATION

We evaluate the model by testing it on a large corpus of JavaScript code mined from open source repositories, over 269K lines of code. The experiments are run on a 2.5GHz machine with 62G RAM.

4.1 Research Questions

By addressing the following research questions we investigate how effective the deep learning model is at identifying bugs and offering repair suggestions. Figure 4 is an overview of our evaluation process.
RQ1 How long does it take to train a model and what hyper parameter values maximize the BLEU score?
RQ2 Is the approach able to learn the differences between buggy and correct code?
RQ3 How helpful are the fix suggestions?

4.2 Datasets

Open source repositories were mined for the `$.ajax` bug pattern, shown in Listing 1, along with its corresponding fix. Change pairs were then formed and processed using the method mentioned in section 3.

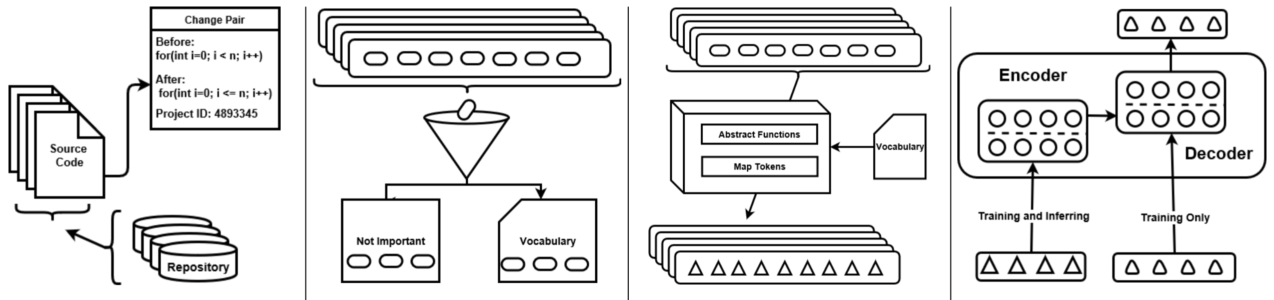
The train and development set consist of nominal (3) and mutant repair (2) change types, the test set consists of nominal and repair(1) change types. Change pairs associated to a given project are grouped to the same data set. For example, if change pairs 1, 2, 3 belong to Project A then all three changes will be found in the train, development or test set. A breakdown of the distribution of change types in each set is shown in Table 2.

Dataset Stats			
Name	Nominal	Repair	Mutant Repair
Train	23153	0	1553
Development	2547	0	182
Test	62	42	11

Table 2: Dataset Change Type Statistics

4.3 RQ1: Hyperparameter Tuning

Hyperparameter tuning is crucial for achieving a high performance encoder decoder model. Hyperparameters affect the model's ability to learn the features that distinguish between buggy and correct code. The number of hidden layers and units per hidden layer are adjusted simultaneously. For the number of hidden layers we experimented with small discrete values; 2,3,4. For each hidden layer configuration we adjusted the number of units in each layer to 64, 128 and 256. We found that a model with 128 units, 2 layers, 5000 training iterations and a learning rate of 0.001 with the Adam optimizer resulted in a high BLUE score for the development set (92.3) while keeping the training time low. The dropout value, which prevents overfitting, was kept constant at 0.2. Training this model took about one hour.



(a) Mine Change Pairs (b) Build Vocabulary (c) Abstract Out Tokens (d) Produce Repair Suggestion

Figure 3: Overview of proposed Encoder Decoder approach for repair suggestion.

4.4 RQ2: Bug Detection Accuracy

To measure bug detection quality we use three metrics: Precision (Equation 3), Recall (Equation 4) and F1 Measure (Equation 5), which are widely used in the field of software defect detection [9, 21]. The precision metric shows us how many of the identified bugs were actually bugs. The recall metric shows us how many bugs the model was able to discover from the actual set of bugs. The F1 measure is a combination of both the precision and recall metrics. To measure repair suggestion quality we use the Bilingual Evaluation Understudy (BLEU) score which is shown to be correlated with human judgement [16]. The BLEU score is a value between 0-100, the higher the score, the more the useful the generated repair is for developers.

A True Positive (TP) occurs when the inputted sequence should be formatted as a JSON object and the model correctly inserts `JSON.stringify` around the data field. A False Positive (FP) occurs when the inputted sequence is not defective and the model outputs a changed sequence (e.g drastic change; change in operation, incorrect insertion of `JSON.stringify`). A True Negative (TN) occurs when the inputted sequence is not defective and the model does not make any changes to the code. A False Negative (FN) occurs when the inputted sequence should be formatted as a JSON object and the model does not insert `JSON.stringify` around the data field.

$$Precision = \frac{True\ \ Positive}{True\ \ Positive + False\ \ Positive} \tag{3}$$

$$Recall = \frac{True\ \ Positive}{True\ \ Positive + False\ \ Negative} \tag{4}$$

$$F1 = 2 * \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5}$$

The encoder decoder model is trained on the train dataset and evaluated on the test dataset. During evaluation, Precision, Recall, F1 Measure and BLEU score metrics are recorded, Table 3 shows the performance of the model.

4.4.1 Precision. The precision metric, calcualted using Equation 3, shows us how many of the reported bugs were actual bugs. Table 4 summarizes our findings. High precision scores indicate that most bug alerts are actual bugs and helps with developer adoption since less false alarms will be raised. A high precision for nominal tells us that the model will preserve correct code.

Performance Statistics				
Type	TP	FP	TN	FN
Repair	6	3	31	2
Mutant	5	3	3	0
Nominal	1	2	59	0
All	12	8	93	2

Table 3: Performance of model on datasets

Precision Scores	
Type	Precision
Repair	0.67
Mutant	0.63
Nominal	0.97
All	0.6

Table 4: Precision Scores

4.4.2 Recall. The recall metric, calcuated using Equation 4, shows us how many bugs were identified from the total set of bugs. Table 5 summarizes our findings. High recall scores indicate that the model is able to identify most bugs. The model has higher mutant and nominal recall scores since the training set consists of these change types. Improving recall for real repairs requires a tradeoff in terms of precision, more alerts can be broadcasted but it may result in more false alarms.

Recall Scores	
Type	Recall
Repair	0.75
Mutant	1
Nominal	1
All	0.86

Table 5: Recall Scores

4.4.3 F1 Measure. F1, calcuated using Equation 5, measures the accuracy of our bug detection by combining precision and recall scores. Table 6 summarizes our findings.

F1 Measure Scores	
Type	F1 Measure
Repair	0.71
Mutant	0.77
Nominal	0.98
All	0.71

Table 6: F1 Measure Scores

4.4.4 BLEU Score. The BLEU score measures how close the generated repairs are to actual repairs, scores closer to 100 correlate more to a human suggestion. Table 7 summarizes our findings. High BLEU scores on both the development and test set indicate that the model's repair suggestions are close developer repair suggestions.

BLEU Scores	
Dataset	BLEU Score
Development	92.3
Test	84.0

Table 7: BLEU Scores

4.5 RQ3: Repair Accuracy

The BLEU score is a good heuristic to measure repair suggestion quality but not adequate for our process. Abstracting input sequences (Section 3.3) increases the degree of token mismatch since tokens that exist in the actual correct sequence may not be in the vocabulary list generated. The model is not able to output the exact value for these tokens but rather it is abstracted type (Table 1) or the `<unk>` token. The BLEU score penalizes suggestions that contain abstract types even if they are semantically correct.

Take 7, and 8 where the parameter `c` has been abstracted into `<abstractToken>`, a abstract token type (Table 1). The two sequences have different string literals but the exact same semantic meaning, we consider the sequences equal. To validate the suggestion quality, taking semantic equivalence into account, our model's repair suggestions are classified manually before computing the precision and recall scores.

Listing 7: Actual Correct Code

```
result = method(a, b, c)
```

Listing 8: Inferred Correct Code

```
result = method(a, b, <unk>)
result = method(a, b, <abstractToken>)
```

We refer to Table 3 for quality metrics. 6 out of 8 bug instances had the correct repair suggestion and only 3 out of 42 examples had `JSON.stringify` incorrectly inserted. 31 of the repair examples did not specify `application/json` as the contentType and therefore were unchanged by the model, this shows us that the model was able to learn the relationship between `contentType` and `data`. These observations give us confidence in the model's effectiveness to detect the majority of bugs, recognize correct code and offer suggestions at a low false postive count.

5 DISCUSSION

This section will discuss the implications, limitations, and threats to validity of the results. It also outlines suggestions for future work.

5.1 Implications

5.1.1 Developers. Our approach helps JavaScript developers write clean code quickly by providing them with early and actionable alerts. The low false positive count, good precision and high recall in Section 4.4 and 4.5 shows developers that the model's alerts can be trusted.

5.1.2 Companies. Companies can integrate this tool into the developer workflow at the code check in stage as a preventative measure to prevent common bug patterns from entering their code bases. Companies can also take counteractive measures by fixing existing buggy code by automatically inserting suggested repairs on source files.

5.1.3 Automating Software Engineering Tasks. Application of our code learning approach is not restricted to only code repair. Contextual understanding of software code supports many other important software engineering tasks such as code suggestion, code search and test case generation which can be explored.

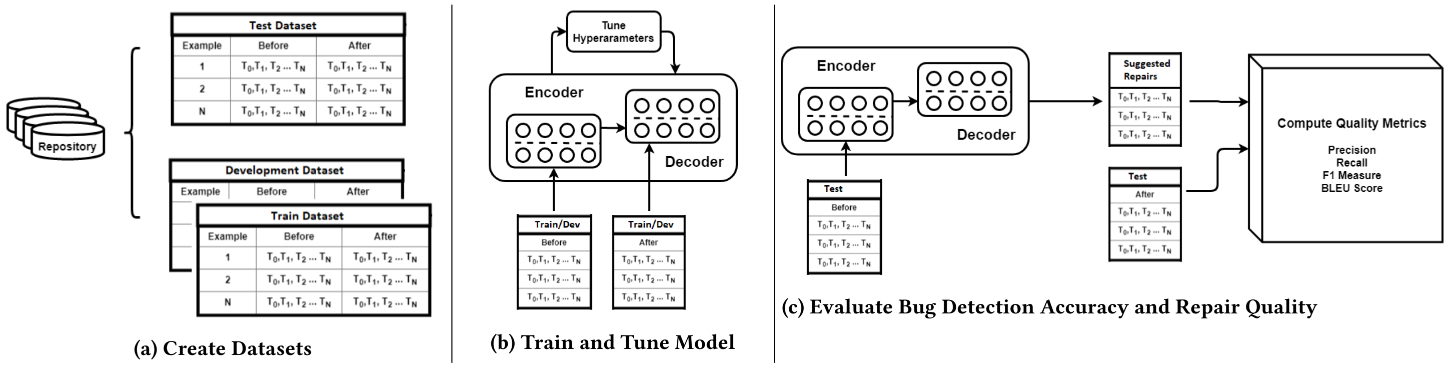
5.2 Limitations

The language model we use abstracts code details such as nested function context and identifier names, therefore some language constructs are not modeled and reamin unlearned. The model will not be able to provide useful repair suggestions when it encounters an unknown language construct. A better language model with finer granularity, one that models nested function context or preserves the developer specified identifier name, may be able to detect more bug patterns and offer more detailed repair suggestions. However, we believe our model includes the most prevalent and relevant JavaScript constructs which give us enough context to offer useful suggestions.

Our approach requires massive amounts of training data to recognize repairs for bug patterns, we assume we can mine a large number of bug patterns and fixes from open source repositories. Although additional examples can be generated through mutating correct code, training on common and actual bugs produced by developers is preferred. Another concern is the integrity of the mined examples, our approach relies on random developers to write correct code. We assume that the after sequence in a change pair is always correct, however, this is not always the case. Referring to Table 3 there is an interesting observation for the Nominal change type, it has one TP. Further analysis on this example shows that the after sequence was incorrect, however, given the buggy input the model was able to produce a useful repair suggestion. It is interesting to note that the model was able to fix incorrect examples in the dataset.

5.3 Threats to Validity

Internal Validity Our model was not trained on real repairs but rather mutant repairs, real repairs were used to evaluate the repair suggestion quality, it is possible that we may acheive better results by training on real repairs. However, Table 6 shows that mutant



(a) Create Datasets

(b) Train and Tune Model

(c) Evaluate Bug Detection Accuracy and Repair Quality

Figure 4: Overview of our approach for evaluating our bug detection and repair model.

repairs are a good substitute for real repairs, the similar scores between mutant repairs and real repairs indicates that generated bug examples are a good representation of the actual bugs found in the real world for this specific pattern.

External Validity An advantage to mining open source JavaScript repositories is that we're able to capture a variety of different programming styles (eg. ES6, ES7, Vanilla JS). It is unknown how effective the model is on a closed source repository that may contain company specific rules, technologies and styles. Examples of company specific rules need to be mined or manually added to the train dataset. It is also unclear how the model will perform on different languages as code structure and AST tokens vary from language to langauge.

5.4 Future Work

Additional bug patterns can be provided to evaluate the model's generizability and ability to suggest repairs for multiple bug patterns. Some things to explore are whether or not the model is (1) able to differentiate and repair bug patterns from different APIs (2) transfer learned patterns between functions that have similar context. For example, are the rules learned for `$.ajax POST` on [`contentType, data`] applied to `$.ajax PUT` given that they have similar context.

It is also interesting to see if the model can prioritize and select from similar bug patterns. Let's imagine we're looking at the `$.ajax PUT` and `$.ajax POST` functions. If there are two functions with similar syntax but slightly different semantic meanings is the model able to learn the relationships that differentate the two? Can the model learn that for `$.ajax PUT` the tokens [`url, contentType, data`] have a relationship where the object identifier needs to be specificed in the `url` field?

Another idea to explore is vocabulary embeddings (similar to word2vec) that better encode JavaScript language features like reserved words and code conventions. An embedding model encodes words into vectors such that words with similar semantic meaning end up closer to one another in vector space. This could help the model learn similarities between code tokens with similar semantic meaning (e.g while, for). Following this line of thought it might be useful to explore using different neural network structures that can better utilize structural information. For example, Tree Structured LSTMs [20] learn the relationships between parent and child nodes within a tree which can be applied to a program's AST to provide a better structural encoding.

6 CONCLUSIONS

In this thesis we propose a deep learning, sequence to sequence, based approach for automatically learning bug pattern rules and suggesting repairs for JavaScript code. Our approach captures structural and semantic information from a program's AST, processes the ASTs into code token sequences and then feeds these sequences into our encoder decoder model. Our experimental results show a BLEU score of 84.0, precision of 60% and recall of 86% which indicate that the proposed approach detects bugs accurately and provides useful repair suggestions to developers. Our work demonstrates the effectiveness of deep learning in defect detection and repair suggestion, it is one step closer towards automatic defect repair.

REFERENCES

[1] 2016. Software Fail Watch: 2016 in Review. (Jan 2016). https://www.tricentis.com/resource-assets/software-fail-watch-2016/

[2] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation.* 14–23. https://doi.org/10.1109/SCAM.2012.28

[3] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Experiences Using Static Analysis to Find Bugs. *IEEE Software* 25 (2008), 22–29. Special issue on software development tools, September/October (25:5).

[4] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 470–481. https://doi.org/10.1109/SANER.2016.105

[5] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[6] Chris Colah. 2017. Understanding LSTM Networks. (2017). http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[7] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent Neural Network Grammars. *CoRR* abs/1602.07776 (2016). arXiv:1602.07776 http://arxiv.org/abs/1602.07776

[8] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering Bug Patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016).* ACM, New York, NY, USA, 144–156. https://doi.org/10.1145/2950290.2950308

[9] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. 2015. An Empirical Study on Software Defect Prediction with a Simplified Metric Set. *Inf. Softw. Technol.* 59, C (March 2015), 170–190. https://doi.org/10.1016/j.infsof.2014.11.006

[10] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don&#039;T Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877

[11] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. 2013. Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *International Conference on Software Engineering (ICSE)*.

[12] Minh-Thang Luong, Eugene Brevdo, and Rui Zhao. 2017. Neural Machine Translation (seq2seq) Tutorial. *https://github.com/tensorflow/nmt* (2017).

[13] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches. *Automated Software Engg.* 17, 4 (Dec. 2010), 375–407. https://doi.org/10.1007/s10515-010-0069-5

[14] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing. *CoRR* abs/1409.5718 (2014). arXiv:1409.5718 http://arxiv.org/abs/1409.5718

[15] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083143

[16] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL '02)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[17] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. *CoRR* abs/1704.07535 (2017). arXiv:1704.07535 http://arxiv.org/abs/1704.07535

[18] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *International Conference on Software Engineering (ICSE)*.

[19] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. 2014. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. *CoRR* abs/1402.1128 (2014). arXiv:1402.1128 http://arxiv.org/abs/1402.1128

[20] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *CoRR* abs/1503.00075 (2015). arXiv:1503.00075 http://arxiv.org/abs/1503.00075

[21] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 297–308. https://doi.org/10.1145/2884781.2884804

[22] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ÅĄukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). http://arxiv.org/abs/1609.08144