A RECURRENT NEURAL NETWORK BASED PATCH RECOMMENDER FOR LINUX KERNEL BUGS

by

Anusha Anand Bableshwar

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical and Computer Engineering

Charlotte

2019

Approved by:

Dr. Arun Ravindran

Dr. Hamed Tabkhi

Dr. Andrew Willis

ABSTRACT

ANUSHA ANAND BABLESHWAR.
A Recurrent Neural Network Based Patch Recommender for Linux Kernel Bugs.
(Under the direction of DR. ARUN RAVINDRAN)

Software bugs result in a variety of issues including system crashes, loss of system performance, incorrect output and security vulnerabilities. In this thesis, we explore the design of a patch recommender system for the Linux kernel. Software tools that aid the developer in quickly developing bug fixes can help in improving programmer productivity. Previous efforts in automated bug resolution uses a search approach for exploring the automatically generated patch space for functionally correct patches. In contrast, we focus on bug reports, and patch commit descriptions manually generated by humans and expressed in a natural language such as English. Our goal is to relate a new bug description to the most closely related patch that resolved a similar bug in the past. Compared to existing approaches, we do not attempt to generate a patch code, but instead point the user to potential patches that enable developers to identify the part of the code base that they should focus on. Our approach is thus complementary to existing research. We explore the use of Natural Language Processing (NLP) to mine bug/patch descriptions. Our dataset consists of previously resolved bugs and the corresponding patches from the Linux kernel project. We pose the bug-patch matching as a semantic similarity NLP problem. After generating a custom word embedding for the bug-patch dataset, we train a Siamese LSTM network that outputs the Manhattan distance between bug and the patch. The Keras-Tensorflow framework is used. We then evaluate our approach with bugs from the test set, and determine the top-K matches for the bug from all existing patches. At the 50^{th} percentile of the test bugs, the correct patch occurs within top 11.5 patch recommendations output by the model.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor, Dr Arun Ravindran for giving me the opportunity, guidance and support to purse and succesfully complete my thesis in one of the most absorbing research topics. He provided immense expertise at every stage. He is one of the most versatile mentors who allowed me to explore distinct ideas and steered me in the right direction.

I would like to express my gratitude to my committee members, Dr. Hamed Tabkhi and Dr. Andrew Willis for their valuable feedback. I would also like to thank our research collaborator Mr. Manoj Iyer from Canonical, the company behind the Ubuntu Linux distribution, for introducing us to the research problem addressed in the thesis and providing guidance regarding the Linux kernel. My earnest appreciation to the University of North Carolina at Charlotte for supplying me with essential infrastructure and helping me through the entire process.

This would not have been possible without constant encouragement from my friends Sai Amrit Bulusu, Aishwarya Gunasekar, and my family. Especially my parents, who made this a smooth sailing journey. And of course a gigantic "thank you" to all my fellow researchers and well wishers.

DEDICATION

dedicated to my parents Shobha Bableshwar and Anand Bableshwar.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

BRNN Bi-Directional Recurrent Neural Network.

FFNN Feed Forward Neural Network.

LSTM Long Short Term Memory.

MaLSTM Manhattan LSTM

NLP Natural Language Processing

RNN Recurrent Neural Network.

CHAPTER 1: INTRODUCTION

Software bugs result in a variety of issues including system crashes, loss of system performance, and incorrect output. More ominously, bugs result in software vulnerabilities that are exploited by attackers to launch a variety of cyberattacks against computer systems. While progress has been made in developing automated tooling for static analysis of source code, and automated patching of bugs, bug mitigation still remains a manually intensive job. For complex projects such as the Linux kernel, the majority of the bugs are largely detected by users, and reported using bug tracker systems such as Bugzilla[1]. In response to the reported bug, developers release code patches to fix the bug. Due to the large volume of bugs reported (for example, hundreds per day for the Linux kernel), and the limited availability of developer time to fix these, many of these bugs remain unresolved for a considerable length of time. Unfortunately, more the number of days since Day Zero (that is, when the bug was first reported) that the bug continues to exist, the greater the vulnerability of it being exploited by attackers to compromise the system.

In this thesis, we explore the design of a patch recommender system for the Linux kernel. Software tools that aid the developer in quickly developing bug fixes can help in improving programmer productivity, as well as ensure that bugs are fixed quickly. In general, bug resolution is a complex task often involving deep understanding of the code base, and the underlying computing system. Nevertheless, in many cases similarities exist between new bugs and previously resolved bugs. In such cases, software patches that have been developed for previously resolved bugs may provide valuable clues to the developer to fix newly reported bugs.

We investigate if we can possibly exploit information from existing bugs and the

corresponding patches (bug-patch pair) to aid developers in fixing new bugs. Previous efforts in automated bug resolution have focused on the bug and the patch code. Automated bug resolution is cast as a search problem exploring the automatically generated patch space for functionally correct patches. In contrast, we focus on bug reports, and patch commit descriptions manually generated by humans and expressed in a natural language such as English. Our goal is to relate a new bug description to the most closely related patch that solved a similar bug in the past. Compared to existing approaches, we do not attempt to generate a patch code, but instead point the user to potential patches that enable developers to identify the part of the code base that they should focus on. Our approach is thus complementary to existing research.

We note that using a simple keyword search is often insufficient, since the semantic context of the descriptions need to be understood to relate bug reports to patch commits. In this thesis, we explore the use of Natural Language Processin (NLP) to mine bug/patch descriptions. Recent years have seen huge gains in NLP due to the unprecedented success of Recurrent Neural Networks (RNNs). In particular, a family of RNNs known as Long Short-Term Memory (LSTM) networks has been highly successful in NLP tasks such as sentiment analysis, language translation, semantic similarity matching, and text summarization.

1.1 Summary of approach

The focus of our effort is on the Linux kernel owing to the open source nature of the project, and the potential impact due to the widespread use of Linux in all manner of computing systems including embedded, mobile, workstations, cloud and supercomputers. Linux bugs are reported online on Bugzilla. Patches to the bugs are available on the Linux source tree[2]. We scrape the two websites to create a bug/-patch pair data set after suitable pre-processing. We pose the bug-patch matching as a semantic similarity NLP problem. Note that unlike traditional semantic similarity

problems such as detecting duplicate text, the bugs and patch descriptions are fairly dissimilar. After generating a custom word embedding for the bug-patch dataset, we train a Siamese LSTM Recurrent Neural Network based on 70% the original bug-pair data set using the Keras-Tensorflow framework. The data set is augmented with mismatched bug-patch pairs - that is, where the bug and the patch are not related. We then evaluate our approach with bugs from the test set, and determine the top-K matches for the bug from all existing patches. At the 50^{th} percentile of the test bugs, the correct patch occurs within top 11.5 patch recommendations output by the model.

1.2 Key contributions

The thesis makes the following contributions -

- To the best of our knowledge this is the first reported attempt to use NLP on bug and patch text descriptions to build a patch recommender system
- Developed a bug-patch labeled data set for the Linux kernel for use by other researchers
- Generated a custom word embedding for the unique vocabulary of Linux kernel bugs and patches
- Designed a Siamese LSTM based recommender system for predicting closest matching patch for an input kernel bug

1.3 Organization of thesis

The rest of the thesis is organized as follows - Chapter 2 presents a background on Linux kernel bugs, previous reported work related to this research, and deep learning. Chapter 3 presents the Siamese LSTM model used in the recommender system. Chapter 4 describes the data collection methodology, and the generation of custom word embedding. Chapter 5 outlines our evaluation approach and presents the results. Chapter 6 concludes the thesis with a summary of results and future research directions.

CHAPTER 2: BACKGROUND

In this chapter we present a brief background of kernel bugs, LSTM Recurrent Neural Networks, as well as previously reported work related to this research.

2.1 Linux Kernel Debugging

The Linux kernel is currently the most widely used operating system running on a variety of platform including embedded devices, mobile phones, desktops/workstations, cloud servers, and supercomputers. The kernel is an actively maintained open source project with contribution from hundreds of developers worldwide. Bugs in the kernel are reported on kernel Bugzilla by users who encounter bugs. A bug report consists of the relevant Linux subsystem (for example, Drivers), the kernel version the bug was encountered on, the description of the problem, as well as relevant *dmesg* (kernel message buffer) output. Additionally, the bug report has a title which is supposed to succinctly describe the bug. The title and the description of the bug is written in English with relevant jargon used by the kernel development community. The bug is then assigned to a developer associated with the relevant subsystem, who develops a kernel patch that fixes the bug. The developer also includes a succinct title that describes the patch, as well as an English text description of the what the patch fixes. The patch then undergoes extensive testing and is made available on the kernel git tree.

Bug fixing is a time consuming activity depending on subtlety of the bug, on whether the bug is hard to reproduce, and the type of kernel subsystems affected by the bug. Also, many of the patches in the kernel source tree are not bug related, but add new features to the kernel. Often, these new patches might trigger bugs in the

kernel leading to a regression, which can cause a particular subsystem to not function as expected, or worse - result in a kernel OOPS, or kernel panic. Analyzing the bug requires deep understanding of the relevant subsystem, and a series of git bisects to figure out which patch triggered the bug. Since device drivers account for bulk of the Linux kernel today (about 70%), and drivers are typically written by less experienced developers, a variety of bugs are from the driver subsystem. In many cases, similar bugs may have been resolved for bugs reported on similar devices from other vendors.

The example below gives the reader a sense of what a Linux kernel bug and the corresponding patch description looks like -

```
Bug 1526312 - No touchpad - error: i2c_hid i2c-SYNA3602:00:
unexpected HID descriptor bcdVersion (0x00ff)

Reported: 2017-12-15 08:11 UTC by Dietrich
Modified: 2018-12-26 17:55 UTC (History)
CC List: 36 users (show)
Fixed In Version: kernel-4.19.2-301.fc29 kernel-4.19.3-300.fc29
kernel-4.19.3-200.fc2

User-Agent:       Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:57.0)
Gecko/20100101 Firefox/57.0
Build Identifier:
On my newly bought laptop I tried to install Fedora 27
Neither Touchpad nor Touchscreen are working.
The touchscreen "works" with a firmwarefile - but does random things
- so maybe wrong firmware?
For the touchscreen I did choose the firmware from here:
```

https://github.com/onitake/gsl-firmware/blob/master/firmware/onda/
v891w/FW_I89_GSL3676B_19201200_.fw renamed that to: mssl1680.fw

With that firmware the laptop does random things like opening
rightclick menu and closing windows and opening the gnome shell...
It's kind of funny but not usefull :(
As for the touchpad I never got it to do anything useful.
I tried adding kernel parameters: i8042.nomux=1 i8042.reset
according to: https://bbs.archlinux.org/viewtopic.php?id=226212
But not success.
Reproducible: Always
Steps to Reproduce:
1. boot
2. touch
Actual Results:
3. no touch :(
Expected Results:
Having touch features

And the associated patch

author Julian Sax <jsbc@gmx.de>2018-09-19 11:46:23 +0200
committer Jiri Kosina <jkosina@suse.cz>2018-09-29 21:25:59 +0200
commit 9ee3e06610fdb8a601cde59c92089fb6c1deb4aa (patch)
tree 90689de6f079e313896cadd3282c29fad0b70601
parent dc4e05d079591c6f69bb28a07bcc13d4f1c9993b (diff)
download linux-9ee3e06610fdb8a601cde59c92089fb6c1deb4aa.tar.gz
HID: i2c-hid: override HID descriptors for certain devices
A particular touchpad (SIPODEV SP1064) refuses to supply the HID

```
descriptors. This patch provides the framework for overriding these
descriptors based on DMI data. It also includes the descriptors for
said touchpad, which were extracted by listening to the traffic of the
windows filter driver, as well as the DMI data for the laptops known
to use this device.

Relevant Bug: https://bugzilla.redhat.com/show_bug.cgi?id=1526312
```

2.2 Related Work

Automatic bug repair (also known as bug patching) has been receiving a steady stream of attention starting in the mid 2000s. Monperrus [3] has provided a comprehensive review of the literature as of 2018. Bug repair consists of the following elements - (1) a bug oracle that determines the existence of the bug, and (2) a repair strategy that is employed to fix the bug. Further, the repair strategy could be a behavioural repair - where the source code is transformed (compile time or run time), or it could be a state repair - where the state of the system under repair is changed (run time only). Examples of state repair include re-initialization and restart, checkpoint and rollback, input and environment modification.

The work presented in this thesis is a behavioral repair strategy. our goal is to serve as aid to the kernel developer by recommending existing patches that are most closely associated with the bug. We, therefore, focus our review on papers that are most closely aligned to our approach. In general these approaches use test suites and static analysis tools to detect bugs, and then generate patches for these bugs through either known solutions for different bug patterns, or by mining existing patches for possible solutions.

In the BugFix project [4], association rule learning is used to generate a database of relationships between resolved bugs in the code and the corresponding code patches.

Bugs in new code are detected either through failing tests, or by searching the code for known bug patterns. A prioritized list of bug-fix suggestions from the database (based on a confidence value) is then made available to the user. The project is evaluated on the Siemens benchmark of relatively small C programs.

In the Prophet project [5], a parameterized probabilistic model trained from existing patches, is used to rank candidate patches generated for a detected bug. Correct patches are shown to share general features across applications and hence is learnable. The approach is evaluated on large open source application in C that include gzip, python, wireshark, and php.

In the Getafix project [6], Facebook describes a tool that automatically suggests bug fixes for engineers. Existing bugs and patches are mined to create a collection of patch templates considering the broader context of the bug fixes. New bugs found by their static analysis Infer tool are resolved by performing pattern mining using a hiearchial clustering technique and extraction of abstract patterns. Facebook has successfully deployed this tool in production.

Other recent research includes the Deep Repair [7] projects that use Deep Learning analysis of code to generate candidate patches from the same code base, and the SimFix [8] project that uses generates candidate patches from an intersection of existing patches and code snippets in the same project. Both these projects are evaluated on the Defects4J Java benchmark.

Contrary to all the above approaches to automated bug resolution that target the source code of the bugs and patches, in our work we use human generated text description of bugs and patches. Unlike bugs detected by static analysis tools, bugs reported by users are in a text format that describes the symptoms of the bugs. In a complex code base such as the Linux kernel, it is not readily apparent what causes the bug. The goal of our approach is to guide the developer to the right parts of the code base to start the bug patching process. Once the right part of the code

base that is the source of the bug is identified, the approaches outline above can be employed to patch the bug. We, therefore, consider our approach as complementary to the existing research on automated bug patching.

In the Deep Triage project [9], a RNN based-model is used to process bug reports so as to assign it to the appropriate developer. However, unlike our work, no attempt is made to recommend bug patches.

2.3 Deep Learning

Deep Learning is a part of the machine learning family of algorithms which works on data representations as opposed to task specific algorithms. The underlying goal of Deep Learning is to replicate a human brain architecture made of neurons connected in series and parallel as an Artificial neural network. Among the different classes of artificial neural networks, Recurrent Neural Networks (RNN) has had the most impact on the Natural Language Processing (NLP) algorithms we intend to apply to our bug automation problem. The core concepts of Recurrent Neural Networks are laid out in the next section.

Feedforward neural networks, often called as Deep feedforward networks are the standard deep learning models. Neurons are connected in layers with the first layer taking inputs and the last layer producing outputs. All the other layers are hidden from the external world. Each neuron takes weighted sum of inputs applies an activation function and returns an output(Figure : 2.1). The operation of each neuron are as mentioned in equations 2.1 and 2.2

$$Z^T = W^T * X^T \tag{2.1}$$

$$Y^T = A(Z^T) \tag{2.2}$$

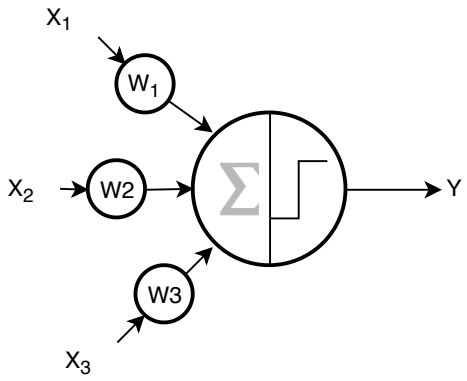

Figure 2.1: Operations of a single Neuron

2.3.1 Recurrent Neural Networks

A Recurrent Neural Network (RNN) [10] is a type of neural network architecture which has a loop pointing to the same circuit (Figure : 2.2) unlike the normal feed-forward neural networks.

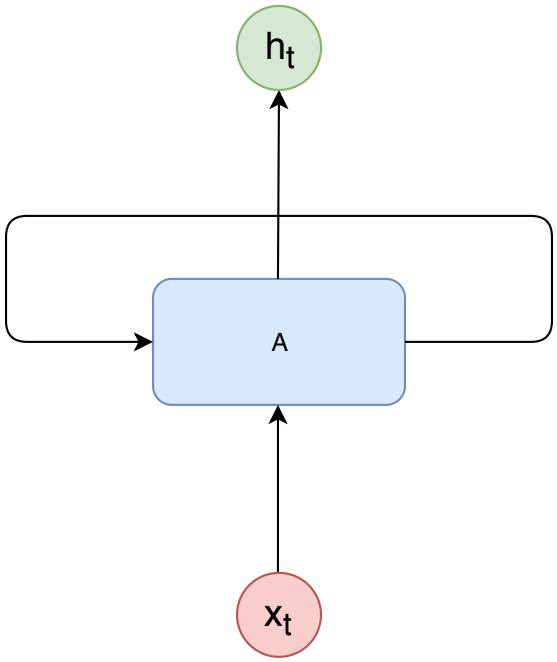

Figure 2.2: A chunk of RNN

Simple RNNs are a network of neurons where each neuron at a time-step (t), receives the input vectors as well as its own output from the previous time-step $(t-1)$.

The output of each neuron for a time-step (t) in an RNN step is a state (y_t), so the input to each neuron of the next step is the previous output state $y_{(t-1)}$ and the input x_t. The operation for each neuron:

$$y_{(t)} = \phi({x_{(t)}}^T.w_x + {y_{(t-1)}}^T.w_y + b) \tag{2.3}$$

Here, w_x is the weight of input and w_y is the weight of the previous output.

RNNs that can take in sequential inputs and give out sequential outputs are called sequence-to-sequence models, such as used in language translation. On the other hand, when the network is fed a sequence of inputs, and ignore all outputs except for the last one, this is called a sequence-to-vector model(Figure:2.3). We have used sequence-to-vector model for our purpose.

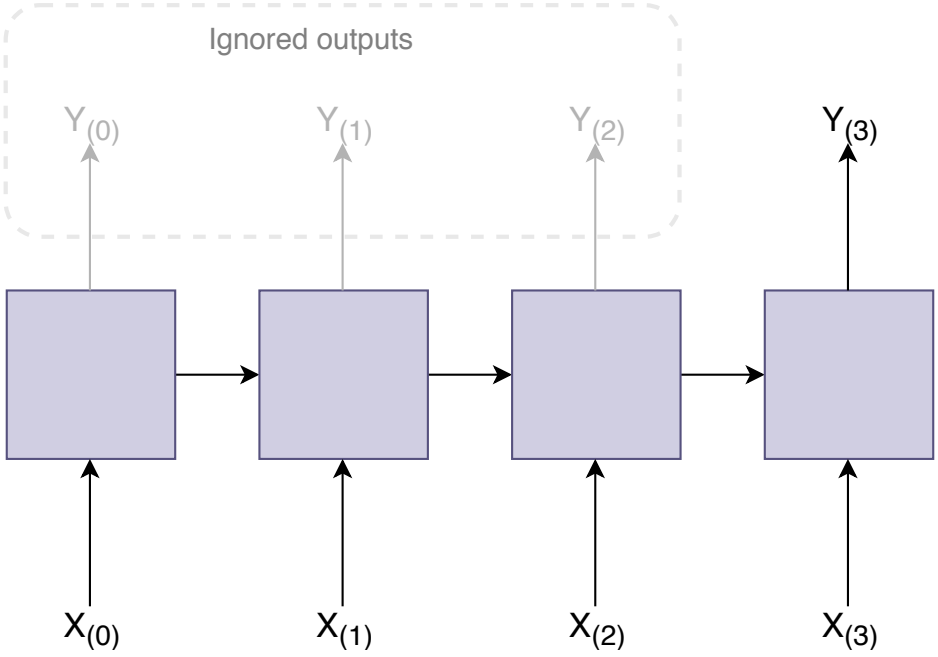

Figure 2.3: Sequence to Vector learning RNN unrolled.

Since an RNN network takes the previous times-step's output and one of the inputs, we can assume that it has some sort of memory retention. Each module of the network is called a *cell*. However, RNNs have trouble with long term dependencies since

the gradient of the loss function decays exponentially leading to vanishing gradient problem. On the other hand, when the error gradients accumulates in very large networks, it leads to exploding gradient problems. A solution to this problem would be to use cells which have the quality of long-term memory.

2.3.2 Long-Short Term Memory (LSTM)

One example of such cells is a Long-Short-Term Memory (LSTM)[10] cell. The LSTM cell was proposed in 1997 by Sepp Hochreiter and Jurgen Schmidhuber [11]. LSTMs can be very useful while dealing with long term dependencies. The most important concept of an LSTM cell is that it has two state vectors and they are kept separate. The architecture of the LSTM cell is shown in Figure 2.4. LSTMs also have a repeating module like the RNN, but the repeating modules have a different structure. Each cell has 4 neural network layers. The key to an LSTM cell is the state vector that runs throughout the cell. Its value can be modified with minor linear interactions. It can easily flow through without any alterations as well. This state vector is changed according to what needs to be remembered and what needs to be forgotten, these operations are regulated by structures called gates. Gates optionally let information through.

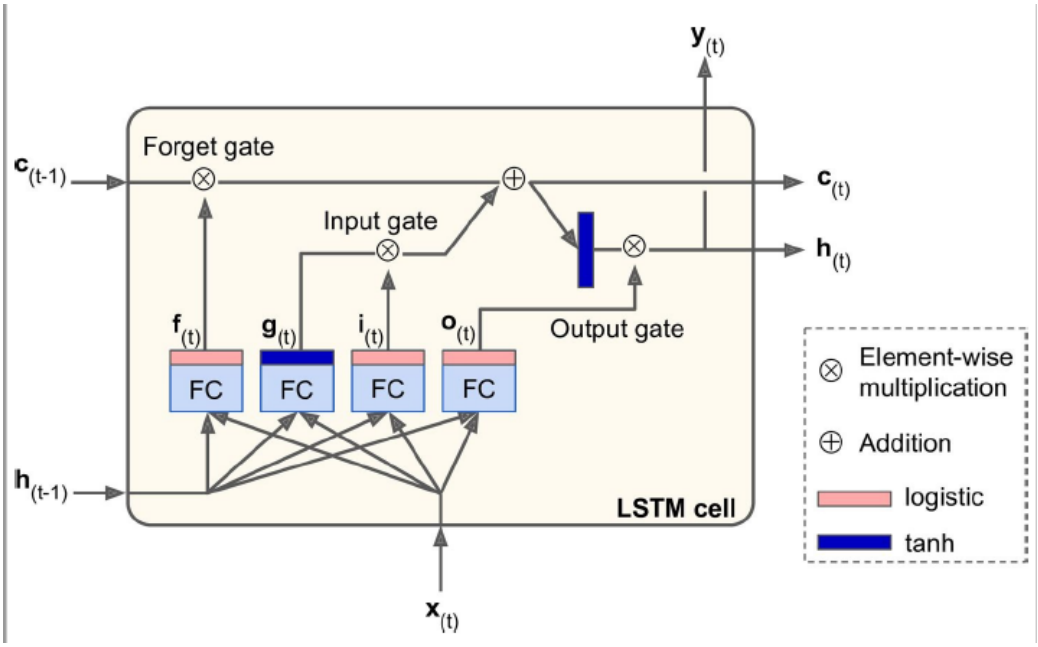

Figure 2.4: LSTM Architecture [10]

The state of an LSTM cell is split into two vectors h and c where h is short-term state and c (c stands for *cell*) is the long-term state. The *forget gate layer* decides what information to throw away or to not retain. It takes in $h_{(t-1)}$ and x_t and gives out either 0 or 1 (Eq: 2.5) for the cell state $C_{(t-1)}$, 0 means forget this state and 1 means remember this state. The next step is to decide what information gets updated to the new one(Eq: 2.6). The *input sigmoid layer* decides that with i_t (Eq: 2.4) and the *tanh layer*. The *tanh* creates a vector $g_{(t)}$, to be (Eq:2.7) added to the cell state. We then update the old state into the new state - we multiply f_t with the old state and then add $i_t * g_{(t)}$ (Eq:2.8). Finally, we need to decide which parts of the long-term state should be read and produced as outputs $y_{(t)}$ (Eq:2.9)which is done by the sigmoid layer. The *tanh layer* is used to limit the outputs between -1 and 1. The LSTM cell has the ability to notice important inputs and store it for as long as needed and extract it when needed. Equations 2.4

$$i_{(t)} = \sigma({W_{xi}}^T.x_{(t)} + {W_{hi}}^T.h_{(t-1)} + b_i) \tag{2.4}$$

$$f_{(t)} = \sigma({W_{xf}}^T.x_{(t)} + {W_{hf}}^T.h_{(t-1)} + b_f) \tag{2.5}$$

$$o_{(t)} = \sigma({W_{xo}}^T.x_{(t)} + {W_{ho}}^T.h_{(t-1)} + b_o) \tag{2.6}$$

$$g_{(t)} = tanh({W_{xg}}^T.x_{(t)} + {W_{hg}}^T.h_{(t-1)} + b_g) \tag{2.7}$$

$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \tag{2.8}$$

$$y_{(t)} = h_{(t)} = o_{(t)} \otimes tanh(c_{(t)}) \tag{2.9}$$

W_{xi}, W_{xf}, W_{xo}, W_{xg} are the weight matrices of each of the four layers for their connection to the input vector $x_{(t)}$. And W_{hi}, W_{hf}, W_{ho}, W_{hg} are the weight matrices of each of four layers for their connection to the previous short-term state $h_{(t-1)}$. And all the biases are present for respective layers.

CHAPTER 3: DESIGN OF THE RECOMMENDER SYSTEM

In this chapter we elaborate on the design of the proposed patch recommender system. Our goal is to relate a new bug description to the most closely related patch that solved a similar bug in the past. We note that using a simple keyword search is often insufficient, since the semantic context of the descriptions need to be understood to relate bug reports to patch commits. We explore the use of NLP to mine bug/patch descriptions formulating the bug-patch matching as a text matching problem. However, different from the typical application of text matching to detect duplicates, we instead use the degree of match as a similarity metric. We leverage the success of RNNs in NLP and base our matching model on a Siamese LSTM model. For a new bug, the recommender system then evaluates the bug against all the patches recommending the top-K patches for the bug.

3.1 Model

We use an MaLSTM model [12] to generate a similarity score between bug and patch descriptions, MaLSTM has a shared LSTM network (Siamese LSTM) with two inputs - the bug description and the patch description. The outputs are the labels that describe the relationship between the bug and the patch (0 or 1). The words in the text are represented using the word embedding matrix (See Section 4.3. The MaLSTM uses the LSTM network to read in word-vectors that represent each input text and employs its final hidden state as a vector representation for each description. The vector representation of these descriptions are used to calculate the Manhattan distance between them. There are 4 layers in the model, the input layer, the embedding layer, the LSTM layer and the Manhattan layer.

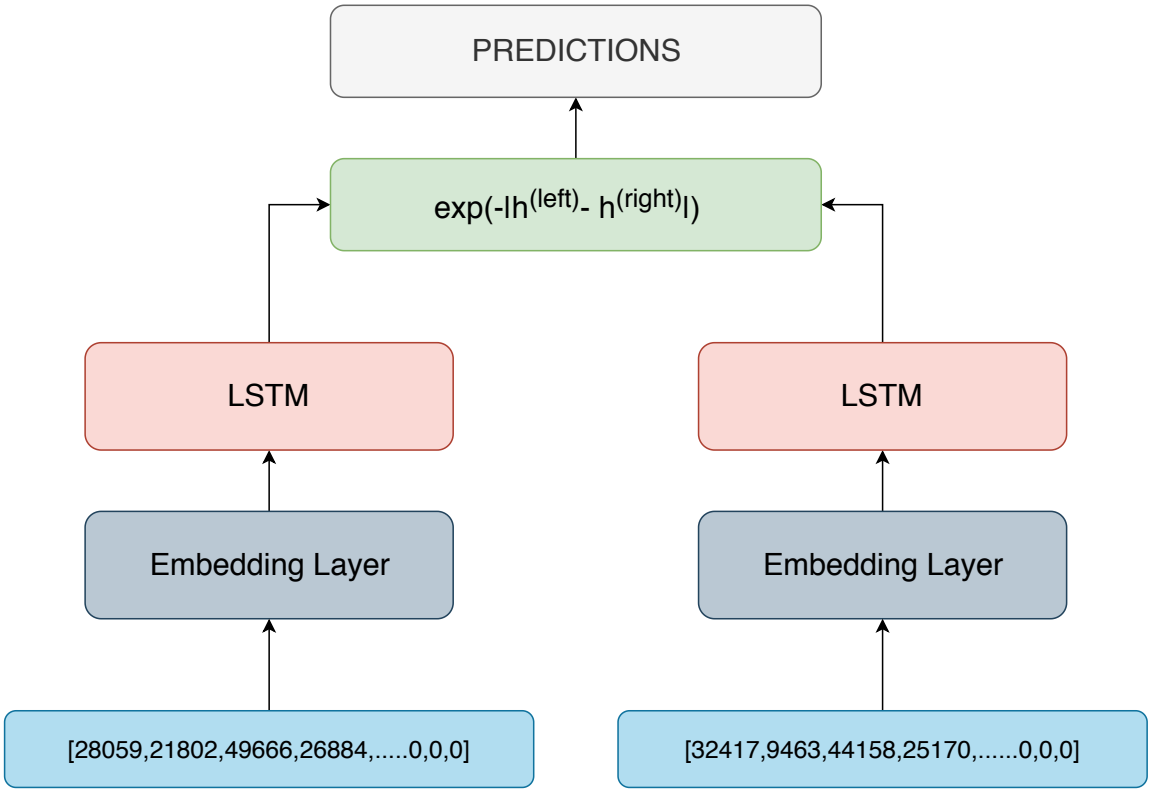

Figure 3.1: The illustration of the MaLSTM model. The sentences are represented using word embeddings in the embedding layer. Outputs from the LSTMs are used to obtain a representation between the bug and that patch before the model predicts.

Figure 3.1 shows the MaLSTM. The boxes with the same color carry the same weight. These layers are shared between the the two inputs and each layer has two nodes to serve each input. The outputs of the LSTM layer are vectors representing the bug and the patch description. These vectors are used to calculate the Manhattan distance using Equation 3.1.

$$y = exp(-|h^{(left)} - h^{(right)}|) \tag{3.1}$$

The model is trained on the bug-patch data and drives the output (Manhattan distance) closer to the target variable by reducing the loss function after each training epoch. We minimize the negative log-likelihood of the observed labels at training time.

$$L(y, y^p) = -y * logy^p + (1 - y) * log(1 - y^p) \tag{3.2}$$

In Equation 3.2 , y is the actual class of the sample and y^p is the predicted Manhattan distance of the class for the sample.

After the model is trained, we use this model to make patch recommendations for the bugs. The model predicts a Manhattan Distance between the bug description and the patch description. The new bug is tested against all the existing patches and Manhattan distances are calculated. The patches are sorted according to their Manhattan distances and top-K patches are chosen from these. For practical purposes, K would be a small number between 5 and 10.

CHAPTER 4: DATA COLLECTION AND PREPARATION

The bug-patch dataset is obtained from two open source platforms (Bugzilla [1] and Linux Source Tree [2]). The data collected is a combination of bug-patch pairs with label 1, and bug-(non-patch) pairs with label 0 (See Figure 4.1). The data collection and processing process and how the data is modified to be fed in to the model are explained in this chapter.

4.1 Data scraping

Webscraping is the automated process of extracting information from the targeted website. For the analysis of the bug and patch, we scrape the short description part of the bug and patch reports. We did this using BeautifulSoup [13] - a Python library for parsing HTML data. Beautiful Soup provides a few convenient techniques for navigating through and searching a parse tree.The first step for information extraction using BeautifulSoup is to find the URL source from where we scrape the information. We extracted a list of URLs for patches from Linux source tree. The URLs for bugs was found in the patch descriptions. To extract a specific section of the page we identify the structure of the the webpage's HTML. After locating the the URLs, writing the code for scraping is fairly simple. The content from the webpage is requested from the server by using the $get()$ method and the response stored in a variable. The content of the page is saved by $.text$. By creating a BeautifulSoup object we can access the data under each tag of the HTML tree.

4.2 Data Pre-processing

In [43]:
```
data.head()
```

Out[43]:

	bug_desc	label	patch_desc	patch_links
0	When using kernel 4.10 or newer DVBSky S960 an...	1	As pointed at:\r\n\thttps://bugzilla.kernel....	https://git.kernel.org/pub/scm/linux/kernel/gi...
1	distribution slackware hardware envi...	0	When the driver notices that PCIe link is gone...	https://git.kernel.org/pub/scm/linux/kernel/gi...
2	distribution debian hardware environmen...	0	Add ELAN0612 to the list of supported touchpad...	https://git.kernel.org/pub/scm/linux/kernel/gi...
3	distribution none but it occurs in suse...	0	A hardware/firmware error may happen at any po...	https://git.kernel.org/pub/scm/linux/kernel/gi...
4	distribution debian sid hardware envir...	0	This reverts commit 55aedef50d4d810670916d9fce...	https://git.kernel.org/pub/scm/linux/kernel/gi...

Figure 4.1: Example of Scraped dataset.The document is divided into bug description, patch description, label followed by their respective patch links

In [6]:
```
(data.patch_desc[50])
```

Out[6]:
```
'ACPICA commit d3c944f2cdc8c7e847b7942b1864f285189f7bce\r\r\n\r\r\nWindows seems to allow arbitrary table signatures for
Load/load_table\r\r\nopcodes:\r\r\n  ACPI BIOS Error (bug): Table has invalid signature [PRAD] (0x44415250)\r\r\nSo this
patch removes dynamic load signature checks. However we need to\r\r\nfind a way to avoid table loading against tables li
ke MADT. This is not\r\r\ncovered by this commit.\r\r\n\r\r\nThis Windows behavior has been validated on link #1. An end
user bug\r\r\nreport can also be found on link #2.\r\r\n\r\r\nThis patch also includes simple cleanup for static load si
gnature check\r\r\ncode. Reported by Ye Xiaolong, Fixed by Lv Zheng.\r\r\n\r\r\nLink: https://github.com/acpica/acpica/c
ommit/d3c944f2\r\r\nLink: https://github.com/acpica/acpica/pull/121 [#1]\r\r\nLink: https://bugzilla.kernel.org/show_bug
.cgi?id=118601 [#2]\r\r\nReported-by: Ye Xiaolong <xiaolong.ye@intel.com>\r\r\nReported-by: Olga Uhina <olga.uhina@gmail
.com>\r\r\nSigned-off-by: Lv Zheng <lv.zheng@intel.com>\r\r\nSigned-off-by: Bob Moore <robert.moore@intel.com>\r\r\nSign
ed-off-by: Rafael J. Wysocki <rafael.j.wysocki@intel.com>\r\r\n'
```

In [7]:
```
(data.bug_desc[50])
```

Out[7]:
```
"Created attachment 216971 [details]\r\ndmesg log\r\n\r\nWe faced with acpi problem in our Supermicro SYS-5027R-WRF serv
er, motherboard X9SRW-F. We have latest BIOS version 3.2a from Supermicro. We tested Debian 8 and CentOS 7 distros with
default kernels. This problem doesn't appear in old 2.6 kernels with CentOS 6 for example. We also tried Debian's 4.x ke
rnel and this problem still exists. Is it critical problem? Or we can just use 3.16 kernels and ignore this messages and
disable these events?\r\n\r\nHere are these two errors:\r\n[    0.410480] ACPI Error: [\\_SB_.PRAD] Namespace lookup fai
lure, AE_NOT_FOUND (20130517/psargs-359)\r\n[    0.410483] ACPI Error: Method parse/execution failed [\\_GPE._L24] (Node
ffff880853ca18c0), AE_NOT_FOUND (20130517/psparse-536)\r\n[    0.410487] ACPI Exception: AE_NOT_FOUND, while evaluating
GPE method [_L24] (20130517/evgpe-579)"
```

Figure 4.2: Example of unprocessed bug and patch descriptions

Pre-processing the data is one of the most important steps before we train the model. As we can see in Figure 4.2, the bug and patch descriptions have a lot of unwanted information that does not contribute the semantic meaning. The first task was to remove all the special characters and symbols. Then all the documents were tokenized and turned to lower case. Stopwords (most commonly occurring words such as a, an, the, they) are removed from the document as they do not add much meaning to the sentence. Python NLTK [14] libraries were used to perform these tasks. The preprocessed data is shown in Figure 4.3.

```
In [166]: data.patch_desc[50]

Out[166]: "['acpica', 'commit', 'cdc', 'bce', 'windows', 'seems', 'allow', 'arbitrary', 'table', 'signatures', 'load', 'opcodes',
          'acpi', 'bios', 'error', 'bug', 'invalid', 'signature', 'prad', 'patch', 'removes', 'dynamic', 'checks', 'however', 'nee
          d', 'find', 'way', 'avoid', 'loading', 'tables', 'like', 'madt', 'covered', 'behavior', 'validated', 'link', 'end', 'use
          r', 'report', 'also', 'found', 'includes', 'simple', 'cleanup', 'static', 'check', 'code', 'reported', 'xiaolong', 'fixe
          d', 'zheng', 'https', 'githubcom', 'pull', 'bugzillakernelorg', 'show', 'bugcgi', 'xiaolongye', 'intelcom', 'olga', 'uhi
          na', 'olgauhina', 'gmailcom', 'signed', 'lvzheng', 'bob', 'moore', 'robertmoore', 'rafael', 'wysocki', 'rafaeljwysocki']
          "

In [165]: data.bug_desc[50]

Out[165]: "['created', 'attachment', 'details', 'dmesg', 'log', 'faced', 'acpi', 'problem', 'supermicro', 'sys', 'wrf', 'server',
          'motherboard', 'srw', 'latest', 'bios', 'version', 'tested', 'debian', 'centos', 'distros', 'default', 'kernels', 'appea
          r', 'old', 'example', 'also', 'tried', 'kernel', 'still', 'exists', 'critical', 'use', 'ignore', 'messages', 'disable',
          'events', 'two', 'errors', 'error', 'prad', 'namespace', 'lookup', 'failure', 'found', 'psargs', 'method', 'parse', 'exe
          cution', 'failed', 'gpe', 'node', 'ffff', 'psparse', 'exception', 'evaluating', 'evgpe']"
```

Figure 4.3: Example preprocessed document of bug and patch descriptions from dataset.

4.3 Word Embeddings

Semantic vectors (also known as word embedding) allows the comparison of semantic meanings of the words numerically. Since there are Linux kernel specific jargon present in our data, a general pre-trained word embedding like GloVe [15] cannot be used. We found that there were only 22% of the words present in Glove from the bug and patch vocabulary. For this research we use word2vec to create our own word embedding to better represent the data. The corpus for training word2vec is generated by picking out all the necessary words by preprocessing the data and removing all the unnecessary words.

Word2vec can be treated as a neural network with a single projection and hidden layer which we train on the corpus. The weights are used as the embeddings, the size of the hidden layer is equal to the dimension of the vector representation. The embeddings can be found in two ways: Skip Gram and Common Bag Of Words (CBOW). CBOW takes the context of each word and tries to predict the context. Whereas Skip Gram takes in the target words and gives out probability distributions for N possible words. CBOW works with large amounts of data and hence suits well for our purpose.

Word2vec groups similar words together in a vector space. Given enough data

word2vec can predict the meaning of words in a context with high accuracy (e.g. "man" is to "boy" what "woman" is to "girl"). These clusters can form the base of sentiment analysis and recommendations. The output of word2vec is a vocabulary for each word having a vector along with it. The dimension of this vector is to be decided by the user; in our experiments we have used 100 as the vector dimension. Figure 4.4 shows an example of words in the corpora similar to the word "error" and their vector representations.

```
In [26]: w1 = "error"
         model.wv.most_similar (positive=w1)

Out[26]: [('warning', 0.43776237964630127),
          ('failure', 0.4295889735221863),
          ('normaly', 0.39054518938064575),
          ('relatives', 0.3902730941772461),
          ('outputting', 0.38914915919303894),
          ('nvidiako', 0.3882807493209839),
          ('data', 0.3866177201271057),
          ('xfffffffffffff', 0.3785271644592285),
          ('printd', 0.37851059436798096),
          ('called', 0.3776209354400635)]
```

Figure 4.4: Example of word2vec similar words

4.4 Data Preparation

Since we cannot feed text words to the first layer of our model, the embedding Layer, we need to convert them to numbers before training. We numerically index the words in the vocabulary starting from 0 to the length of vocabulary. The vocabulary is a list of all the unique words present in the entire text of dataset available. After creating a dictionary, each word as the key and the number assigned as the value, the words in the text are replaced by these integer numbers. Any unknown words appearing here are given an <UNK> token. The integer acts as a key for the words in the embedding matrix and helps us retrieve their vector representations. Both of these (the integer form of words and the embedding matrix) are fed to the embedding layer. The embedding layer replaces the integers with their vectors (n-dimensional) while training and passes these to the MaLSTM layer.

CHAPTER 5: EVALUATION AND RESULTS

In this chapter we describe the evaluation methodology of the proposed patch recommender system, and present results.

5.1 Evaluation

The bug and patch data is collected as described in Chapter 4. The scraped data contains 4963 matching bug-patch pairs and 4899 of bug-patch pairs that do not match. In order to reduce generalization error the data can be augmented to mitigate overfitting problem. By creating more negative data for a bug, we will have more clarity on the match for each bug. To create the additional negative samples we picked 3560 bugs from the matching pairs and paired them against other non-matching patches. Our dataset consists of 27670 samples with a bug having one matching patch and 5 non-matching patches. The dataset is divided into training (19,369), validation (4151) and test (4150). The input data is limited to 100 words per description; those which are shorter are padded with zeroes.

5.1.1 Software and Hardware platform

The implementation of our proposed approach is based on Python programming language working on the Jupyter Notebook [16] platform. Python has a large number of libraries for data processing and machine learning . The framework for the model was implemented using Keras [17] which is based on TensorFlow [18]. TensorFlow is an open source library used for tensor calculations described using data flow graphs. The neural network architectures can be readily be expressed as Tensorflow operations. Keras is a high-level open source neural network library built on top of TensorFlow. The training was done on Nvidia GeForce GTX 1060 GPU with 6 GB

memory. The training for each epoch took about 1 minute and the entire training process of 94 epochs with early stopping took about 1.6 hours.

5.1.2 MaLSTM Model parameters and Evaluation Metrics

The baseline of the MaLSTM model is two parallelly running LSTM layers each with 50 neurons followed by the Manhattan layer which calculates the Manhattan distance between the two vectors. The model is trained using the Adadelta optimizer based on Adagram [19]. Adagrad is an algorithm for gradient-based optimization which performs larger updates for infrequent parameters and smaller updates for frequent parameters. Adadelta is an extension of Adagrad which reduces its aggressiveness, uniformly reducing the learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

The model was trained with a starting learning rate of 0.5, batch size of 64, accuracy threshold of 0.8, and for maximum of 200 epochs with early stopping at 94. Some experiments were done with the embedding layer and observations were made. When the Embedding layer is set to be 'trainable' the validation and training accuracy were seen to be fluctuating a lot and the entire training process was consuming more time. Thus, we decided to keep the Embedding layer to be 'un-trainable'. The Mean-Square Error (MSE) loss is checked after every epoch and if the validation loss does not decrease after four update checks, early stopping of training is done.

The model testing is done by taking a bug-patch matching pair and evaluating the Manhattan distances against all patches (2132 patches) using the MaLSTM model. The patches are then reordered according to the Manhattan distance and the matching patch is located in the list. Our goal is to ensure the presence of the matching patch in the top-k recommended patches, with the lowest possible k.

5.2 Results

We experimented with various numbers and sizes of the LSTM layer and found that 1 LSTM layer with 50 hidden units gave the best results. As seen in Figure 5.1, the train and test accuracy as well as the loss are show a monotonic decrease throughout the training process, indicating that the model is not overfitting over the training data. The model accuracy with the test set after the training of the model was 90.8% after 94 epochs. The accuracy is calculated as:

$$Accuracy = \frac{TruePositive + TrueNegative}{TruePositive + TrueNegative + FalsePositive + FalseNegative} \tag{5.1}$$

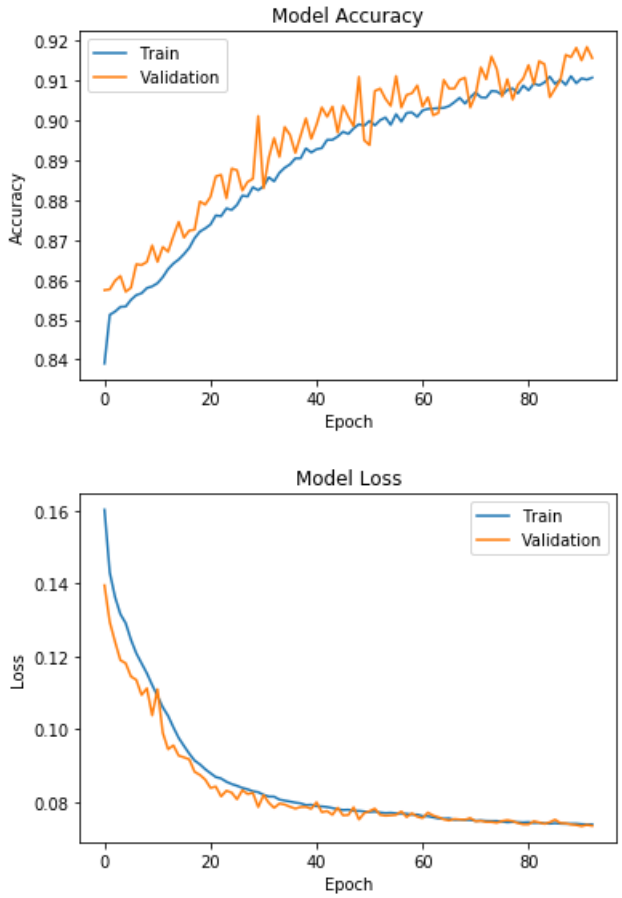

Figure 5.1: Accuracy and Loss of the training and validation set.

However, the goal of this model is not to predict whether the bug and the patch supplied to the model is a match or not. Instead the model serves as a means to calculate a similarity score. Our main purpose of this model is to recommend top-K patches to a new bug. To calculate the performance of the model in that domain, we chose a test bug and calculated the Manhattan distance of that bug against all the patches existing in our database. The Manhattan distance closest to 1 would act as better fixes to the bug than the patches having Manhattan distance closer to 0.

Testing with about 314 bugs, against the 2132 patches, and using a threshold for a valid match, 50% of bugs had the best recommendation (fix patch) made in the top 20 patches.

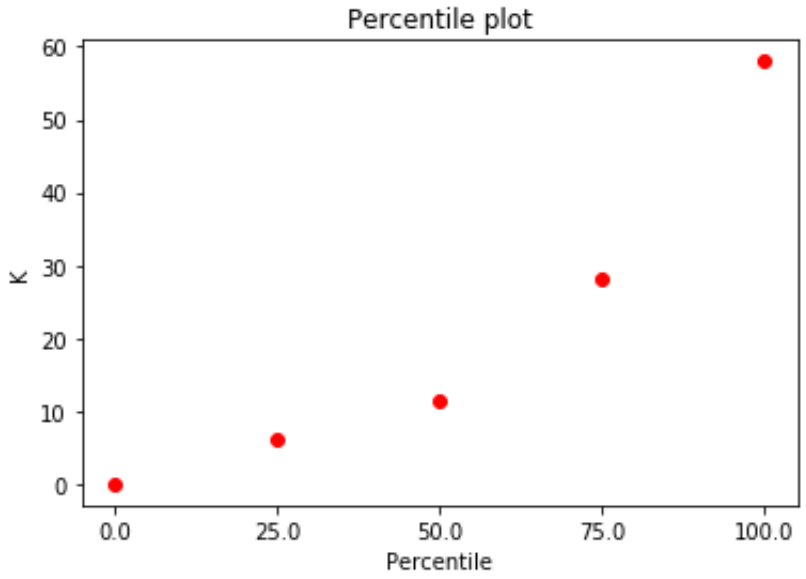

Figure 5.2: Percentile Plot of the Top-K recommendations made

The percentile plot for top K for a 314 bug test set is shown in Figure 5.2. The k value at the 50-th percentile (median) is 11.5, while at the 75-th percentile is 28.25.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

In this thesis we have proposed and demonstrated an alternative approach to patch recommendations for Linux kernel bugs. Our approach uses the natural language description of bugs and patches instead of the established technique of using the bug and patch source code. As such, the proposed approach can be considered as a first step to bug fixing for complex projects such as the Linux kernel. Based on the patch recommendations of our model for a new input bug, the developer will have a better sense of which part of the code base to target. Initial results indicate that at the 50^{th} percentile of new test bugs, the model outputs the correct patch in its top 11.5 recommendations, while at the $75th$ percentile, the model outputs the correct patch in its top 28.25 recommendations.

Future extensions of this work would seek to integrate code along with the text descriptions. The MaLSTM model used in our work could also be used to correlate bug descriptions with patch code. For a new input bug, the model would then be able to recommend patch code to the developer. Another line of research could focus on using encoder-decoder networks to "translate" bug descriptions to patches. A similar approach was used by Jiang, Armally, and McMillan [20] to automatically convert code diffs to commit messages.

REFERENCES

[1] "Bugzilla dataset." https://bugzilla.kernel.org/describecomponents.cgi.

[2] "Linux kernel source tree." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?qt=grepq=.

[3] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, pp. 17:1–17:24, Jan. 2018.

[4] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "Bugfix: A learning-based tool to assist developers in fixing bugs," in *2009 IEEE 17th International Conference on Program Comprehension*, pp. 70–79, IEEE, 2009.

[5] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *ACM SIGPLAN Notices*, vol. 51, pp. 298–312, ACM, 2016.

[6] "Getafix: How Facebook tools learn to fix bugs automatically." https://code.fb.com/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/.

[7] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," *CoRR*, vol. abs/1707.04742, 2017.

[8] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (New York, NY, USA), pp. 298–309, ACM, 2018.

[9] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triaging," in *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pp. 171–179, ACM, 2019.

[10] A. GeÌron, *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2017.

[11] S. Hochreiter, "Long short-term memory," 1997. Neural computation.

[12] J. Mueller and A. Thyagarajan, "Siamese recurrent architectures for learning sentence similarity," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[13] V. G. Nair, *Getting Started with Beautiful Soup*. Packt Publishing, 2014.

[14] E. Loper and S. Bird, "Nltk: the natural language toolkit," *arXiv preprint cs/0205028*, 2002.

[15] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[16] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows.," in *ELPUB*, pp. 87–90, 2016.

[17] F. Chollet, "Keras." https://keras.io, 2015.

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[19] S. Ruder, "An overview of gradient descent optimization algorithms." http://ruder.io/optimizing-gradient-descent/, 2016.

[20] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 135–146, IEEE Press, 2017.

APPENDIX A: KERAS MA-LSTM MODEL

```
n_hidden = 50
gradient_clipping_norm = 1.25
batch_size = 64
n_epoch = 500
left_input = Input(shape=(max_seq_length,), dtype='int32')
right_input = Input(shape=(max_seq_length,), dtype='int32')
embedding_layer = Embedding(output_dim=embedding_dim, input_dim=nb_words
    , weights=[word_embedding_matrix], input_length=max_seq_length,
    trainable=False)
# Embedded version of the inputs
encoded_left = embedding_layer(left_input)
encoded_right = embedding_layer(right_input)
# Since this is a siamese network, both sides share the same LSTM
shared_lstm = LSTM(n_hidden)
left_output = shared_lstm(encoded_left)
right_output = shared_lstm(encoded_right)
# Calculates the distance as defined by the MaLSTM model
malstm_distance =Lambda(function=lambda x:
    exponent_neg_manhattan_distance(x[0], x[1]),output_shape=lambda x: (
    x[0][0], 1))([left_output, right_output])
# Pack it all up into a model
malstm = Model([left_input, right_input], [malstm_distance])
#We need set an optimizer
# Adadelta optimizer, with gradient clipping by norm
optimizer = Adadelta(clipnorm=gradient_clipping_norm, lr=0.5)

#Now we will compile and train the model.
malstm.compile(loss='mean_squared_error', optimizer=optimizer, metrics=[
    threshold_binary_accuracy])
es = EarlyStopping(monitor='threshold_binary_accuracy', mode='auto',
    verbose=1, patience=4)
```

```
malstm_trained = malstm.fit([X_left_train, X_right_train], Label_train,
                            batch_size=batch_size, nb_epoch=n_epoch,
                            validation_data=([X_left_val, X_right_val],
                                Label_val),callbacks=[es])
```

APPENDIX B: DATA PRE-PROCESSING

```
columns = ['patch_desc', 'bug_desc']
for col in (columns):
    #remove special characters
    data[col] = data[col].str.replace('[^a-zA-Z\.]', ' ')
    data[col] = data[col].str.replace(r'[^\w\s]','')
    data[col] = data[col].str.lower() #lower casing all words
    data[col] = data[col].str.replace(' +', ' ')#more than one space
    data[col] = data[col].str.replace('0+', ' ')#more than one 0
    data[col] = data[col].str.replace('\n', ' ')
    data[col].dropna(inplace=True)
    data[col] = data[col].apply(word_tokenize)#tokenize

    #Remove Stopwords (commonly occurring words) from the data
    for i in (range(0,len(data))):
        a = []
        for word in data[col][i]:
            if word not in stop_words:
                if len(word)>2:
                    a.append(word)
        data[col][i] = a
```