Chapter 5

Automatically Identifying Patches for Linux Stable Releases

5.1 Introduction

In Chapter 4, we present a LPU (Learning from Positive and Unlabeled Examples) and SVM (Support Vector Machine) based approach to automatically identify bug fixing patches. This LPU+SVM based approach relies on 55 features extracted from code changes and thousands of word features extracted from commit logs. All the features are defined manually to characterize how likely a patch being a bug fixing patch. However, the manual creation of features might ignore good features that could help developers identify bug fixing patches. In addition, relationships between words are ignored when the LPU+SVM approach considers bag-of-word representation of text.[1] Thus a richer feature representation of a patch that can naturally capture its inherent and relevant properties by considering both its commit log and corresponding code changes is needed.

Inspired by recent applications of deep learning techniques in software engineering area, we propose a Convolution Neural Networks (CNN) based approach to automatically learn features from commit log and code changes inside a patch for

[1] Bag-of-word represents a text as the multi-set of words that appear in it.

bug fixing patch identification. Considering that code is different from natural language content, our CNN-based approach processes code changes separately from the commit log and takes program structure into consideration. The processed code change and commit log are then merged to form a document. Documents generated from a set of training patches are then fed into a Convolution Neural Networks based model. The CNN-based model learns network parameters, as well as a classifier, from training patches. For a new patch, the trained CNNs will map the patch into a set of feature-value pairs and predict whether it is a bug fixing patch by applying the learned classifier on the feature-value pairs. To evaluate the performance of our newly proposed CNN-based approach, we collect a new dataset including 7,793 recent patches from the Linux project, which have been labeled as bug-fixing patches and non bug-fixing patches. Our evaluation shows that the new CNN-based approach can achieve a much better precision, F-measure, and accuracy, compared to the LPU+SVM approach proposed in Section 4.

The main contributions of this work include:

1. We propose a new Convolutional Neural Networks (CNN) based approach to identify bug fixing patches. In order to make CNN work on a patch, we process the code changes and commit log inside a patch separately. For each code change, we extract statement level differences from a commit and then extract an AST level representation of the difference. Only AST node kinds rather than the actual values and identifier names are considered and merged with words appearing in the commit log to form a document as input for CNN training.

2. We evaluate the new CNN-based approach and compare it with our LPU+SVM approach proposed in Chapter 4 on a new dataset, which contains 7,793 recent Linux patches. We perform a 10-fold cross validation on the new dataset, and the experimental results show that the CNN-based approach can achieve a higher precision (a relative improvement of 187%), F-measure (a

relative improvement of 71%), and accuracy (a relative improvement of 32%) compared to the baseline approach. The new approach suffers a small loss in recall though (a relative decrease of 17%).

5.2 Background

In this section, we first present some background information about the maintenance of Linux kernel stable versions, and some challenges that the maintenance of Linux kernel stable versions poses for automation via machine learning. We then present the machine learning technique that we use, Convolutional Neural Networks.

5.2.1 Context

Linux kernel development is carried out according to a hierarchical model, with Linus Torvalds at the root, who has ultimate authority about which patches are accepted into the kernel, and patch authors at the leaves. A patch *author* is anyone who wishes to make a contribution to the kernel, to fix a bug, add a new functionality, or improve the coding style. Authors submit their patches by email to *maintainers*, who commit the changes to their git trees and submit pull requests up the hierarchy. In this paper, we are most concerned with the maintainers, who have the responsibility of assessing the correctness and usefulness of the patches that they receive. Part of this responsibility involves determining whether a patch is stable-relevant, and annotating it accordingly.

The Linux kernel provides a number of guidelines to help maintainers determine whether a patch should be annotated for propagation to stable kernels. These are summarized as follows (slightly condensed for space reasons):[2]

- It must be obviously correct and tested.
- It cannot be bigger than 100 lines, with context.

[2]Documentation/process/stable-kernel-rules.rst

- It must fix only one thing.
- It must fix a real bug that bothers people.
- It must fix a problem that causes a build error, an oops, a hang, data corruption, a real security issue, or some "oh, that's not good" issue.
- Serious issues as reported by a user of a distribution kernel may also be considered if they fix a notable performance or interactivity issue.
- New device IDs and quirks are also accepted.
- No "theoretical race condition" issues.
- It cannot contain any "trivial" fixes.
- It must follow the submittingpatches rules.
- It or an equivalent fix must already exist in Linus' tree (upstream).

These criteria may be simple, but are open to interpretation. For example, even the criterion about patch size, which seems completely unambiguous, is only satisfied by 93% of the patches applied to the stable versions based on Linux v3.0 to v4.7, as of April 16, 2017,[3] with other patches ranging up to 2754 change and context lines. More generally, different developers may have different strategies for choosing and propagating patches to stable kernels. Figure 5.1 shows the rate of propagation of patches from the various subsystems, where a subsystems is approximated as a subdirectory of `drivers` (device drivers), `arch` (architecture specific support), or `fs` (file systems), or any other toplevel subdirectory of the Linux kernel source tree. While the median rate is 6%, for some subsystems the rate is much higher, raising the possibility that the median is too low and some stable-relevant patches are getting overlooked. If this is the case, part of the problem may be the stable propagation strategies of the individual maintainers. Indeed, as shown in Figure

[3] Duplicates are possible, as a single patch may be applied to multiple stable versions.

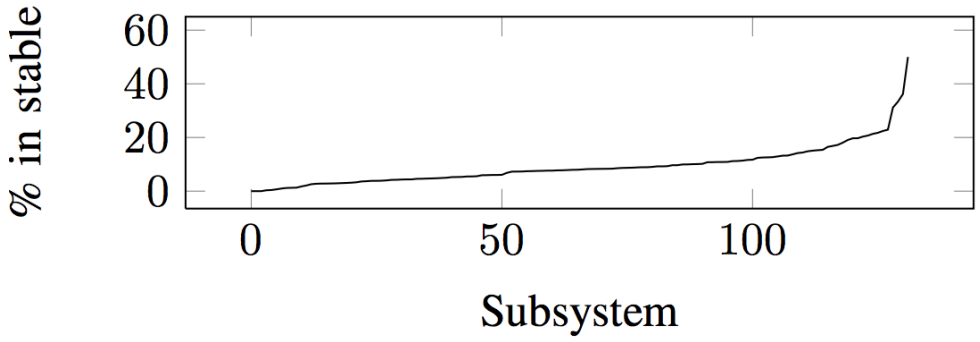

Figure 5.1: Rate at which the patches applied to a given subsystem end up in a stable kernel. Subsystems are ordered by increasing propagation rate.

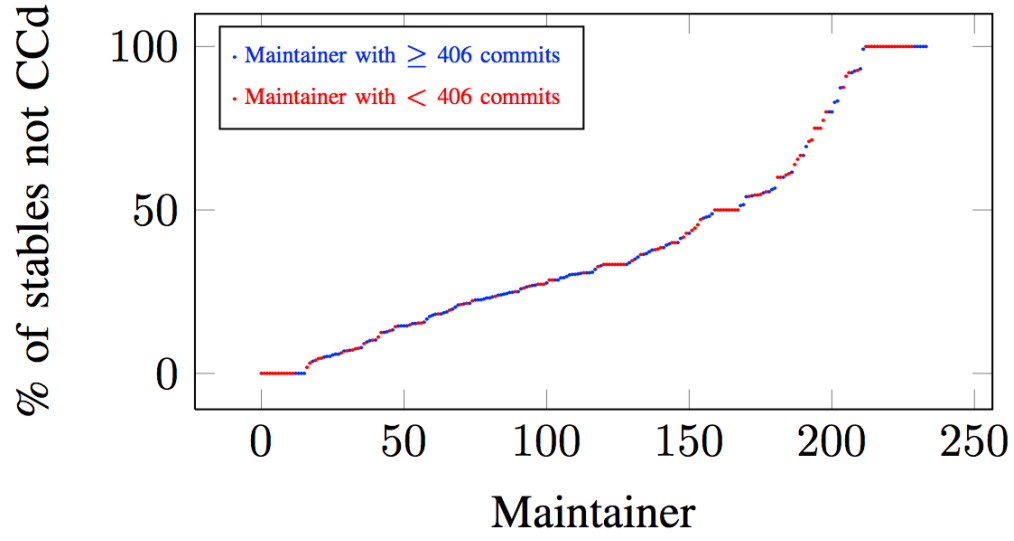

Figure 5.2: Rate at which a maintainer's commits that end up in a stable kernel are annotated with Cc stable. 406 is the median number of commits per maintainer. Maintainers are ordered by increasing Cc stable rate.

5.2, the rate at which a given maintainer's commits that end up in a stable version are annotated with Cc stable covers the full range from 0-100%. While alternative submission options, *e.g.*, via email or via a pull request in the case of networking code, are listed in the stable kernel documentation, Cc: stable is advantageous because it is uniform and thus easy for developers to create tools against.

5.2.2 Challenges for Machine Learning

Stable patch identification poses some unique challenges for machine learning. These include the kind of information available in a Linux kernel patch and the

diversity in the reasons why patches are or are not selected for application to stable kernels.

First, Linux kernel commit logs are written free-form. While maintainers are asked to add a `Cc: stable@vger.kernel.org` tag to commits that should be propagated to stable versions, our goal is to identify stable-relevant commits for which adding this tag has been overlooked, and thus we ignore this information. Patches also contain a combination of text, represented by the commit log, and code, represented by the enumeration of the changed lines. Code is structured differently than text, and thus we need to choose a representation that enables the machine learning algorithm to detect relevant properties.

Second, there are some patches that are applied to stable kernels that are not bug-fixing patches. The stable documentation itself stipulates that patches adding new device identifiers are also acceptable. Such patches represent a very simple form of new functionality, implemented as an array element initialization, but they are allowed in stable kernels because they are unlikely to cause problems for users of stable kernels and may enable the use of the stable kernel with new device variants. These patches have a common structure and are easily recognized, and thus should not pose a significant challenge for machine learning. Another reason that a non-bug fixing patch may be introduced into a stable kernel is that a subsequent bug fixing patch depends on it. These non-bug fixing patches, which typically perform refactorings, should satisfy the criteria of being small and obviously correct, but may have other properties that differ from those of bug-fixing patches. They may thus introduce apparent inconsistency into the machine learning process.

Finally, some patches may perform bug fixes, but may not be propagated to stable. One reason is that some parts of the code change so rapidly that the patch does not apply cleanly to any stable version. Another reason is that the bug was introduced since the most recent mainline release, and thus does not appear in any stable version.

As the decision of whether to apply a patch to a stable kernel depends in part

on factors external to the patch itself, we cannot hope to achieve a perfect solution based on applying machine learning to patches alone. Still, stable-kernel maintainers have reported to us that they are able to check likely stable-relevant patches quickly (*e.g.*, 32 in around 20 minutes).[4] Therefore, we believe that we can effectively complement existing practice by orienting stable-kernel maintainers towards likely stable-relevant commits that they may have overlooked, even though the above issues introduce the risk of some false negatives and false positives.

5.2.3 Convolutional Neural Networks for Sentence Classification

In this paper, we leverage convolutional neural networks (CNN) to automatically learn features for stable bug fixing patch identification. In this section, we present background knowledge about how CNN is applied to automatically learn features for sentence classification tasks. We look at sentence classification tasks because the stable patch identification task could be modeled as a sentence classification task where each sentence contains all the information inside a patch.

In recent years, research and application of neural network based models have grown dramatically. Such models have achieved remarkable results in areas such as computer vision [35], speech recognition [20], and natural language processing (NLP) [64]. Many types of neural networks have been proposed, including Deep Belief Networks (DBN), Recurrent Neural Networks (RNN), Recursive Neural Networks (RNN), Convolutional Neural Networks (CNN), etc. In this work, we consider CNN, which utilize layers with convolving filters that are applied to local features [43]. CNN were originally designed for computer vision, and then have subsequently been shown to be effective for traditional NLP tasks, such as query retrieval [79], sentence modeling [29], and many more. Besides CNN's effectiveness in representing textual information, CNN are usually easier to train and have many

[4]Greg Kroah Hartman, private communication, April 28, 2017

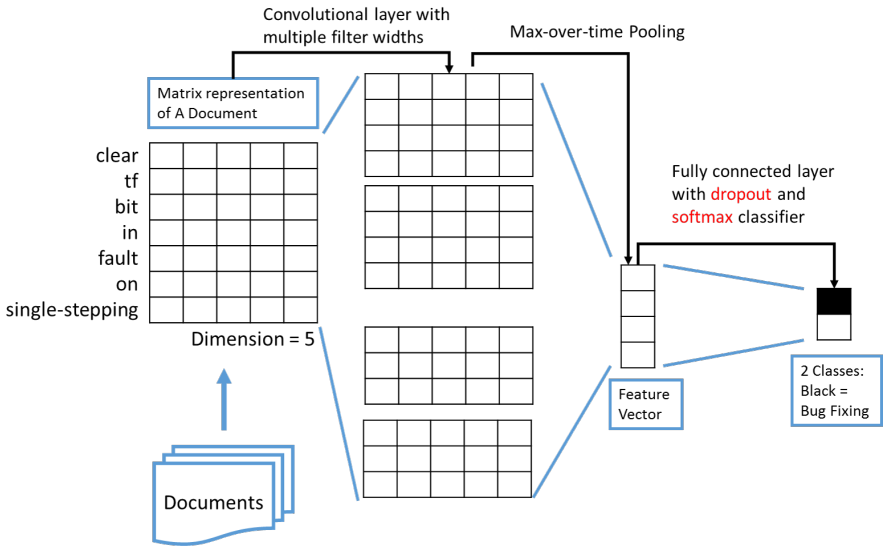

Figure 5.3: Convolutional Neural Networks for Sentence Classification

fewer parameters than fully connected networks with the same number of hidden units.[5]

Figure 5.3 shows the architecture of the Convolutional Neural Network used in our approach. As shown in the figure, the input layer is a sentence (e.g, "clear tf bit in fault on single-stepping") comprised of concatenated word embeddings (i.e., representations of words using vectors). Each word is mapped to a vector of a fixed length (e.g., 5 in Figure 5.3). The input layer is followed by a convolutional layer with multiple filters, then a max-pooling layer, and finally a softmax classifier. This architecture is as the same as the CNN model proposed by Kim for sentence classification [33], except that it learns the word embeddings from the dataset itself rather than using any other pre-trained word vectors, based our hypothesis that word embeddings trained on any other domain or task might not be applicable to this software engineering specific task.

Convolution Layer & Max Pooling: Let $\mathbf{x}_i \in \mathbb{R}^k$ be the k-dimensional word vector corresponding to the i-th word in the sentence. A sentence of length n is represented as

$$\mathbf{x}_{1:n} = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \ldots \oplus \mathbf{x}_n$$

[5]http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

where $\oplus$ is the concatenation operator. Sentences are padded such that they all have the same length as the maximum length sentence in the document. More generally, $\mathbf{x}_{i:i+j}$ refers to the concatenation of words $\mathbf{x}_i, \mathbf{x}_{i+1}, ..., \mathbf{x}_{i+j}$. A convolution operation involves a filter $w \in \mathbb{R}^{hk}$ that is applied to a window of h words to produce a new feature. For example, a feature c_i is generated from a window of words $\mathbf{x}_{i:i+h-1}$ by

$$c_i = f(\mathbf{w} \cdot \mathbf{x}_{i:i+h-1} + b)$$

where $b \in \mathbb{R}$ is a bias term and f is a non-linear function, e.g., the hyperbolic tangent, i.e., $tanh$ in this CNN. This filter is applied to each possible window of words in the sentence $\{\mathbf{x}_{1:h}, \mathbf{x}_{2:h+2}, \ldots \mathbf{x}_{nh+1:n}\}$ to produce a feature map

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

with $c \in \mathbb{R}^{nh+1}$. Next, a max pooling operation [11] is applied over the feature map and takes the maximum value $\hat{c} = max\{c\}$ as the feature corresponding to this particular filter. The max pooling operation is designed to capture the most important feature, *i.e.*, the one with the highest value, for each feature map.

The above process describes how a feature is extracted from one filter. CNN uses multiple filters (with varying window sizes) to obtain multiple features. These features then form the penultimate layer and are passed to a fully connected softmax layer whose output is the probability distribution over two classes (i.e., stable-relevant and non stable-relevant fixing).

Regularization: A critical problem when applying machine learning algorithms is how to make algorithms consistently perform well on both training data and new data (testing data). Many methods have been proposed to modify algorithms to reduce their generalization error but not their training error. These methods are known collectively as regularization. The CNN considered in this paper adopts a regularization method called dropout, to prevent overfitting the learned neural

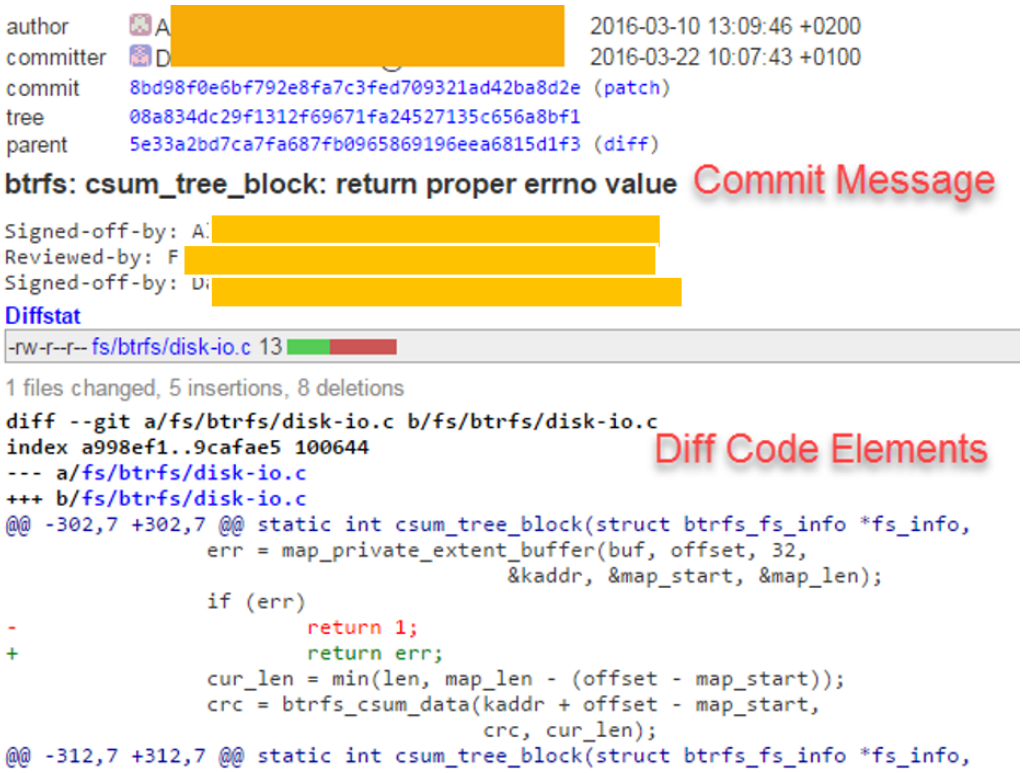

Figure 5.4: Sample Bug Fixing Patch

network to the training data. A dropout layer stochastically disables a fraction of its neurons, which prevents neurons from co-adapting and forces them to individually learn useful features [23].

5.3 Approach

5.3.1 Our Approach

In this work, we leverage the model introduced in Section 5.2.3 to generate features and learn a classification model from the training patches. Figure 5.4 shows a sample patch that has been applied to the stable version derived from Linux v4.5 as the patch fixes user-visible bugs and improves error handling and stability. As illustrated in Figure 5.4, a patch contains not only a textual commit message but also a set of diff code elements, i.e., changes that are applied on the buggy file.

Figure 5.5 illustrates the framework of our approach. Our approach is composed

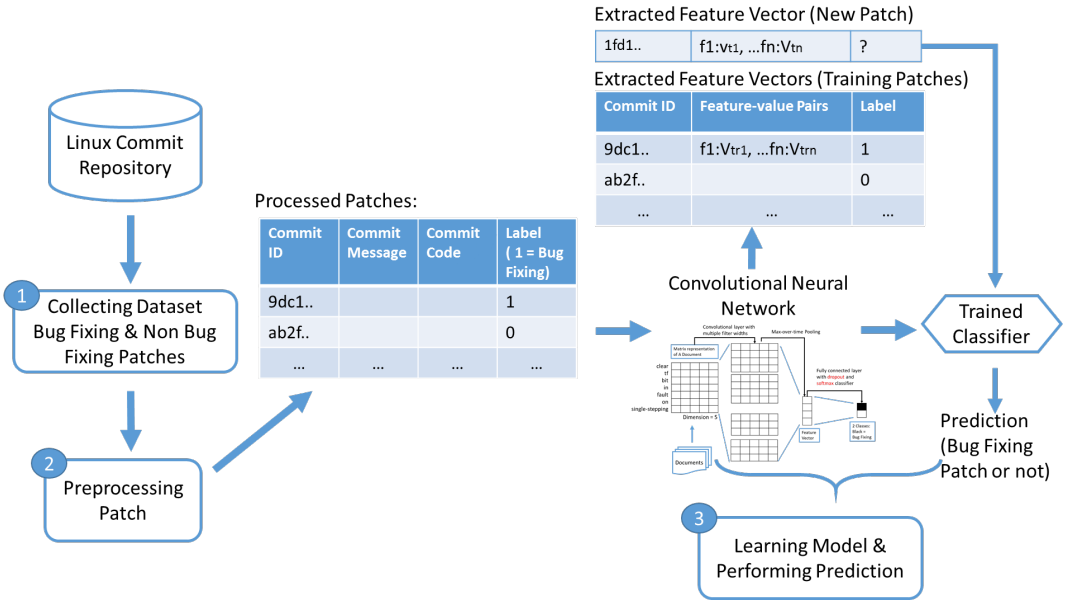

Figure 5.5: Framework of Convolutional Neural Network Based Stable-Relevant Patch Identification

of three steps: collecting the dataset, processing patches, and learning the model & performing prediction. In the first step, we collect a set of recent stable and non-stable patches from the commit repository of the Linux kernel, and annotate them as the stable-relevant patches (i.e., positive instances) and the non stable-relevant patches (i.e., negative instances), respectively. In the second step, we transform each collected patch into a document that a Convolutional Neural Network could take as input. Note that as code is different from natural language, we process the commit message and the diff code elements separately and merge them into one document (see "Processed Patches" in Figure 5.5). In the last step, features/representations of training patches as well as a classifier (based on the learned features) are trained. The trained Convolutional Neural Network is able to convert any new patch into a set of feature-value vectors (see "Extracted Feature Vector (New Patch)" in Figure 5.5) and make a prediction by applying the trained classifier. We elaborate the details of each step in the rest of this section.

5.3.2 Collecting the Data Set

Similar to work presented in Chapter 4, we need to collect a set of stable patches (bug-fixing patches) as well as a set of non stable patches. For each patch, we collect the following information: commit id, author name, date on which the author provided the patch, committer name, date on which the committer committed the patch, the subject line, back link information (stable patches only), number of lines of changed code and context code (by default, typically the three lines before and after each *hunk* of contiguous removed and added lines). A back link is the reference to the same patch in the mainline Linux kernel. Next, we describe in more detail how we collect the data set.

1) Collecting Stable Patches: The main challenge in the processing of stable patches is to link them to the corresponding patches in the mainline. Indeed, all patches accepted into a stable version must have previously been applied in the mainline. Some stable patches contain an explicit link back to the corresponding mainline commit. For others, we rely on the author name and the subject line. Subject lines typically contain information about both the change made and the name of the file or directory in which the change is made, and thus should be relatively unique. Accordingly, the collection of stable patches also collects a mapping of back link information such as a fixes tag to the commit identifier that contains the link and a mapping of a pair of an author email address and a patch subject to a commit identifier.

Following the stable patch rules, presented in Section 5.2.1, we keep only stable patches having at most 100 lines of changed and context code. Discarding the larger patches means that our approach may not be able to learn to recognize them, but we consider that such patches are anomalous, and treat special situations that may not generalize.

2) Collecting Mainline Patches: The collection of mainline patches is essentially the same as the collection of stable patches, except that no back links are collected.

As for the stable patches, we limit mainline patches to those of at most 100 lines, to prevent the CNN from creating a model based only on patch size. A mainline patch is recognized as being a stable patch if a stable patch has the same author and subject as the mainline patch or if there is a back link in the stable patch to the mainline patch, according to the mappings collected during the stable patch collection process.

3) Constructing the training and testing datasets: From the set of mainline patches, we collect three sets of patches: 1) the complete set of stable patches, 2) a set of patches that are not recognized as being in any stable version, referred to as *non-stable*, to be used for training, and 3) a set of non-stable patches that are to be used for testing. For the first set, we use the complete set of stable patches for training, to have the most possible information to learn from. As our initial experiments with CNN showed that training works best when the data is balanced, we then extract the same number of non-stable patches for the second set as there are stable patches available for the first set. Finally, as our motivation is to help stable maintainers identify stable-relevant patches that would otherwise be overlooked, our testing data contains only non-stable patches as well. In addition to fitting with our objectives, this strategy has the benefit of allowing us to use the entire set of stable patches in the training data. We take a statistically significant subset of the set of non-stable patches of size at most 100 lines of changed and context code.

5.3.3 Patch Preprocessing

In this step, our approach takes patches collected in the first step as input and processes each one into a document. Each document contains a sequence of tokens that represent the patch. As mentioned before, our approach treats code changes separately from the commit log and combines them in the end to form a document. We describe in detail the methodology below.

1) Extract Atomic Statement Level Difference: Diff code elements may have

many shapes and sizes - a single word, part of a line, an entire line, multiple lines, multiple lines separated by unchanged code, etc. To describe changes in terms of meaningful syntactic units and in particular to provide some context for very small changes, we collect differences at roughly the granularity of atomic statements. These may be, *e.g.*, simple assignment statements, but also if headers, for-loop headers, function headers, etc. We also distinguish changes in error checking code (code to detect whether an error has occurred) and in error handling code (code to clean up after an error has occurred) from changes in other code. Error checking code and error handling code are indeed very common in the Linux kernel, which must be robust, but are disjoint in structure and purpose from the implementation of the main functionality.

For a given commit, the first step is to extract the names of the affected files and to extract the state of those files before and after the commit. For each before and after file instance, we remove comments (taking care to preserve line numbers) and the contents of strings, as changes in comments and within strings are not likely to be stable-relevant. For a given pair of before and after files, we then compute the difference using the command "git diff -U0 old new". For each $-$ or $+$ line in the diff output, we then collect a record indicating the sign, the hunk number, the line number in the old or new version, respectively, and the starting and ending columns of the non-space changes on the line.

The previous step gives the differences, but the granularity may be below that of our atomic statements. For example, if a function call extends over multiple lines, the change could be in a single argument, on a line by itself. We thus then work on the old or new file individually, to map the changed lines to their enclosing atomic statements, as defined above. This process is performed using Coccinelle [60]. It is limited to the set of patterns supported by the Coccinelle script, and fails, causing the patch to be ignored, if there is any changed token that is not taken into account by these patterns or if Coccinelle is not able to parse the code.

As an example of the processing of code changes, consider the code snippets

shown in Figure 5.6. In the before code, the `if` test expression is found to intersect with a changed line, so part of the result is the information about the if header, i.e., `if (x < 0)`. The `return` statement is also found to intersect with a changed line, so another part of the result is `return -1;`. Similar information is obtained for the after code. Due to the return in the if branch, the changed if headers are annotated as coming from error checking code, and the changed return statements are annotated as coming from error handling code. All of these changes are additionally annotated as coming from the same hunk.

Before:	After:
`if (x<0)`	`if (y<0)`
`return -1;`	`return -2;`

Figure 5.6: Code Example

2) Combining Statement Differences into a Code Representation: As a result of this phase, each hunk is represented as a sequence of tokens for the removed atomic statements followed by a sequence of tokens for the added atomic statements, at most one of which can be empty. We could simply concatenate these. To obtain a more precise view of the changes, we instead compute a word-level diff of the two sequences, using the command `git diff --word-diff=porcelain`, where the option porcelain produces the result in a format that eases subsequent processing. The result is a sequence of context (unchanged) tokens, intersprinkled with word-level hunks containing sequences of removed and added tokens. Rather than using word diff, we could alternatively have used tree differencing [**?**] to obtain fine-grained differences that would respect the programming language's syntactic structure. Word diff, however, is faster than tree differencing, because there is no need for parsing the source code, and thus we use word diff in the current approach.

In the result, we could treat the tokens in the diff code elements of a patch like words in the commit message. For example, Figure 5.4 could be treated as a document: "—a/fs/btrfs/disk-io.c +++ b/fs/btrfs/disk-io.c @@ -302,7..." How-

ever, developers may chose unique identifiers, such as "db1200_mmc_led", "db1200_mmc0_dev", "au1200_lcd_res", "au1200_lcd_dev", across different files and functions, even when the identifiers act similarly in the source code. Thus, if we consider all the tokens appearing in the diff code elements, the vocabulary size will be very large. The data will also be very sparse, because these identifiers might appear very few times across the data set. Thus, the extra information will provide little benefit for the learning process. To address this issue, the preprocessing of a patch ultimately drops the specific names of all identifiers, instead representing them all as a single "Ident" token.

3) Combine Code Representation with Commit Log: For each processed patch, we then combine the processed commit message and the diff code elements into a one-line document (with the symbol "##" as the line separator) so that the Convolutional Neural Network introduced in Section 5.2.3 can process it. The words in the representation of the commit log are encoded such that they are disjoint from the words found in the code. We follow the standard methodology in Natural Language Processing (NLP) to extract token sequences from the patch document [Is this referring to the commit log?]. Specifically, we use the *VocabularyProcessor* object from TFLearn [6] to map documents to sequences of word ids [identifiers?].

[It was stated that the words in the code are processed to ensure that they are different than the words in the commit, but this is not necessary because they are already given different numbers.]

5.3.4 Learning Model & Performing Identification

In the last step, documents preprocessed from the training commits are used as input for training Convolutional Neural Networks. To monitor how the trained model general [I don't understand "general"] fits into [why "into"?] the data, we further split out a part of the data from the training data set to form a validation set. When

[6] A deep learning library featuring a higher level API for Tensorflow http://tflearn.org/.

we train a model from the rest of training data, we periodically test the performance (i.e., F-measure of the positive class - bug fixing patches) of the latest model on the validation set. We stop training when we observe that a stable result is achieved on the validation set [is this clear enough?]. We then use the saved trained model to predict labels for unseen patches.

5.4 Evaluation & Results Analysis

In this section, we first describe the dataset that is collected for the evaluation. Next, we describe the experimental settings for our CNN-based approach and the baseline approach. In the end, we describe our evaluation methodology, evalution metric, and present our experimental results.

5.4.1 Dataset

To evaluate our approach, we target mainline versions 3.0 to 4.7. The stable versions build on mainline versions 3.0 to 4.6.[7] Given the fact that there are many more non stable patches existing in the mainline, for the training and evaluation purpose, we randomly collect a significant sample set from all non stable patches. For training, we have collected 16,265 stable patches, and a randomly sampled 14,688 non-stable patches. For evaluation, we again collected another randomly sampled 17,967 non-stable patches. In total, we considered 48,921 patches from Linux. All selected patches have at most 100 lines of change and context code, as stipulated in the stable kernel rules (see Section 5.2.1).

5.4.2 Model Settings

As training of Convolutional Neural Networks usually requires parameter tuning, we take 10% of each training data to form a validation set for parameter tuning,

[7]The stable version building on *e.g.*, mainline version 4.6 is maintained in parallel with the preparation for version 4.7, and potentially onward.

while the remaining 90% are used to learn the parameters of networks [It is not clear what is the difference between tuning and learning. Which happens first? Intuitively, it might seems that one learns first, and then tunes, but they are not presented in that order, so perhaps that intuition is not correct.]. Our experiment code is written in Python and built on top of the TensorFlow Python library [1].

A Convolutional Neural Network contains multiple parameters. In this experiment, each token in the document that is processed from a patch is embedded into a vector of length 16. We consider filters of two different window size, i.e., 4 and 5. The number of filters is set to 32. During training, the dropout ratio is set to 0.5, which means that 50% of the units will be dropped out randomly during training. [How did we come up with these values? Are they standard? are these parameters or hyperparameters?]

5.4.3 Baseline Approach

We take the approach proposed by Tian et al. as the baseline approach [**?**]. Tian et al. proposed a LPU (Learning from Positive and Unlabeled Examples)+SVM (Support Vector Machine) based approach. The LPU+SVM based approach extracts features from both code changes and commit logs that can potentially distinguish bug fixing patches from regular commits. The features are pre-defined by the authors. In contrast, the CNN-based approach presented in this paper constructs features autonomously [is this correct?].

5.4.4 Evaluation Methodology & Metrics

Evaluation Methodology: For the CNN-based approach, we separate the training data into two parts, 90% of them are randomly picked for training a CNN, while the 10% are used as the validation set, to train the best model settings [Is this the 90-10 split that was already discussed? Are settings the same as parameters or hyperparameters?]. For the baseline approach, we reimplemented the LPU+SVM

approach proposed by Tian et al. by treating the stable patches as positive cases, and the non-stable patches as unlabeled cases. Then, both of the trained models built from the training data are applied to the testing data.

We evaluate the performance of a stable patch identification approach on the testing data, which is a significant sample set that is randomly selected from non-stable patches. To avoid the bias that may be introduced by self labeling, we ask the Linux kernel stable version maintainers, Greg Kroah-Hartman and Sasha Levin, to help us evaluate the results. To save their time in labeling, we prepare two set of data for them to label:

Option 1 For each approach, we rank all the testing patches (currently not in the stable tree) based on their probability of being stable, i.e., top patches are considered by the classifier to be most likely to be stable. We then take the top-50 patches for each approach and create a set that covers all these patches. We sent these patches to Greg Kroah-Hartman to label.

Option 2 We randomly select 100 patches that are currently not applied to any considered stable version and send them to Sasha Levin to label.

Evaluation Metrics: Our goal in evaluation option 1 is to compare the ground truth labels provided by the maintainers with the ranking of the patches within the top 50 results produced by each classifier. We thus use two common ranking-based evaluation metrics that evaluate the quality of a ranked list:

- **Accuracy@N**: this metric calculates the ratio of real stable patches in the top-N list provided by each approach.
- **Average precision (AP)@N**: The average precision of a ranked list of potential stable patches is computed as:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{\textit{number of stable patches}}$$

where k is a rank in the returned ranked patches, M is the number of ranked methods, and $pos(k)$ indicates whether the kth patch is stable or not. $P(k)$ is the precision at a given top k methods and is computed as follows:

$$P(k) = \frac{\#\textit{stable patches top k}}{k}$$

- **Normalized Discounted Cumulative Gain (NDCG)@N** NDCG measures the performance of a recommendation system based on the graded relevance of the recommended entities. It varies from 0.0 to 1.0, with 1.0 representing the ideal ranking of the entities. This metric is commonly used in information retrieval and to evaluate the performance of web search engines.

For evaluation option 2, we calculate commonly used evaluation metrics for classification tasks, i.e., precision, recall, and F-measure, for both approaches.

5.4.5 Evaluation Results & Analysis

Applying applying the CNN-model and the baseline on the testing data (all non-stable), the CNN-model identified 4,710 (26.2%) of them to be stable, while the baseline identified 4,341 (24.2%) of them to be stable.

Evaluation option 1: After we sort all the testing non stable patches according to their probability of being stable for both approaches, we find that there is only one commit that overlaps between the top-50 lists of the two approaches.[8] With his labels, Greg Kroah-Hartman noted "Overall, very nice work in finding lots of patches that are relevant."[9]

The Accuracy@N and AP@N of our approach as compared to those of the keyword-based approach are shown in Table 5.1. We notice that the recall of the new CNN-based approach is comparable with that of the baseline approach (reduce

[8] 5c2e08231b68a3c8082716a7ed4e972dde406e4a
[9] Private communication, May 12, 2017.

16% relatively[10]). On the other hand, the CNN-based approach improves the precision and F-measure of baseline by 187% and 71% respectively.[I think that the previous sentence is out of date] We could also observe that the CNN-based approach improves the accuracy of the baseline from 72% to 95%, which is a relative improvement of 32%.[This one too]

Table 5.1: Accuracy@N, Average Precision (AP)@N, Normalized Discounted Cumulative Gain (NDCG)@N: CNN vs LPU+SVM

Approach	**Acc@10**	**Acc@20**	**Acc@30**	**Acc@40**	**Acc@50**
LPU+SVM	0.9	0.85	0.9	0.9	0.9
CNN-based	1	0.9	0.83	0.825	0.84
Difference	**+11%**	**+5.9%**	-7.8%	-8.3%	-6.7%
	AP@10	**AP@20**	**AP@30**	**AP@40**	**AP@50**
LPU+SVM	0.976	0.925	0.909	0.909	0.909
CNN-based	1	0.979	0.948	0.922	0.905
Difference	**+2.4%**	**+5.8%**	**+4.3%**	**+1.4%**	-0.4%
	NDCG@10	**NDCG@20**	**NDCG@30**	**NDCG@40**	**NDCG@50**
LPU+SVM	0.99	0.98	0.975	0.975	0.975
CNN-based	1	0.995	0.988	0.982	0.979
Difference	**+1%**	**+1.5%**	**+0.7%**	**+32%**	**+0.4%**

Evaluation option 2: The precision, recall and F-measure of our approach as compared to those of the keyword-based approach are shown in Table 5.2. We notice that the recall of the new CNN-based approach is still comparable with that of the baseline approach (reduce 16% relatively[11]). On the other hand, the CNN-based approach improves the precision and F-measure of baseline by 187% and 71% respectively. We could also observe that the CNN-based approach improves the accuracy of the baseline from 72% to 95%, which is a relative improvement of 32%.

Table 5.2: Precision, Recall, F-measure: CNN vs LPU+SVM

Approach	**Recall**	**Precision**	**F-measure**
LPU+SVM based	0.545	0.75	0.631
CNN-based	0.545	0.692	0.610
Difference	0%	-7.7%	-3.3%

[10] i.e., $(0.707 - 0.846)/0.846 = -16\%$
[11] i.e., $(0.707 - 0.846)/0.846 = -16\%$

5.4.6 Potential of Combining CNN-based and LPU+SVM-based Approach

From the performance of the two approaches shown in Table **??**, we find that the two approaches return different results. Thus, we now investigate the results obtained by the two approaches in further detail, to gain insight into the potential of combining them.

5.4.7 Methodology

We assess agreement on correct classification and agreement on error rather than agreement in general. We first match the results from the individual approaches for a given testing data in a dichotomous outcome based on the correct or erroneous outcome of the classification by two approaches. This results in a 2×2 table as shown in Table 5.5. In this table, variable d refers to the number of patches that are misclassified by both approaches. If we divide d by the total number of classified patches, we will get an upper limit of the classification accuracy, which could be considered as a very important indicator since it shows the potential gains in classification accuracy that can be obtained by combining two classifiers. On the other hand, variable a refers to the number of patches, which should not be misclassified in a combination. If we divide a by the total number of patches, the result will give a lower limit of the classification accuracy of any combined classification scheme. Variables b and c provide the relative goodness of the two classifiers. Approaches with fewer patches in the off diagonal cells have little to gain from their combination no matter how successfully it is performed.

As each approach produces a ranked list, we test the correlation between these rankings using Kendall's tau coefficient [citation?]. Kendall's tau coefficient ranges between -1 and 1, with -1 indicating that the rankings are completely different, 0 indicating no correlation, and 1 indicating that the results are correlated. [put the formula?]

Table 5.3: Dichotomous Outcome for CNN and LPU+SVM

(a) #Stable Patches identified by both Classifiers.	2863
(b) #Stable Patches identified only by CNN	1847
(c) #Stable Patches identified only by LPU+SVM	1478
(d) #Stable Patches identified by neither	11,779

Although the top-50 patches overlap in only one case for the two approaches, still both approaches annotate many of the same top-50 patches as stable, with a different ranking. Indeed, out of the 99 unique patches in the two top-50 lists, 86 are actually stable and 79 of these 86 are predicted to stable by both approaches.

Table 5.4: Dichotomous Outcome for CNN and LPU+SVM (Option 1)

(a) #Stable Patches found by both Classifiers.	79
(b) #Stable Patches found only by CNN	6
(c) #Stable Patches found only by LPU+SVM	1

Table 5.5: Dichotomous Outcome for CNN and LPU+SVM (Option 2)

(a) #Patches correctly identified by both	66
(b) #Patches correctly identified only by CNN	9
(c) #Patches correctly identified only by LPU+SVM	11
(d) #Patches wrongly identified by both	14

5.5 Discussion on Wrongly Classified Bug-Fixing Patches

When computing agreement scores in Section **??**, we consider both bug fixing patches, and non bug fixing patches. Based on the fact that identifying the minority class, i.e., bug fixing patch, is harder than identifying the majority class, we now discuss the agreement of both approaches on the bug fixing patches. We observe that 254 out of 1,159 bug fixing patches could only be identified by the baseline, while 92 of them could be identified by the CNN based approach. We also find that 86 (7.4%) of bug fixing patches are missed by both two classifiers. These numbers show potential improvement by combining two approaches.

We then manually check three types of bug fixing patches including:

1) Bug fixing patches that could only be correctly identified by LPU+SVM based approach.

2) Bug fixing patches that could only be correctly identified by CNN based approach.

3) Bug fixing patches that are missed by both approaches.

We find that the baseline approach, i.e., LPU+SVM, is good at capturing bug fixing patches that explicitly contain keywords like "fix", for instance, the commit log from commit with id "b348d7d" upstream mentions: "Fix potential out-of-bounds write to `urb->transfer_buffer` ..." But this commit log also contains some other contents which might make the CNN-based approach identify it as non bug fixing one. On the other side, the CNN-based model could capture bug fixing patches without keywords mentioned in the commit logs. For instance, the commit log from commit with id "6858107" only mentions "Kabylake-H shows up as PCI ID 0xa2f0. We missed adding this earlier with other KBL IDs". For bug fixing patches that are missed in both approaches, we find some short patches, such

as commit with id "e02e588", which summarizes the bug fixing behavior in one sentence: "Instead of calling `vmw_cmd_ok, call vmw_cmd_dx_cid_check` to validate the context id for query commands".

5.6 Threats to Validity

In Chapter 4, we manually checked the labels of 500 sampled patches to establish the ground truth, which might introduce experiment bias. Thus, in this work, we instead regard all the labels already been assigned by the Linux maintainers as the ground truth. However in reality, Linux maintainers might also make mistakes, particularly, they might miss some bug fixing patches. We plan to mitigate such threats by asking some Linux maintainers to label parts of the evaluation data set again in the future.

In Chapter 4, we randomly selected 500 commits and use them to evaluate our prediction model. This time, we try to improve the generalizability of the result by increasing the size of testing set to a full data set which contains 7,793 patches. Similar to the evaluation in Chapter 4, we only investigated patches from the Linux project, although the CNN based approach can be easily applied to identify bug fixing patches in other systems. In the future, we would like to consider more projects.

Selection of evaluation metrics might introduce threats to construct validity. To mitigate such threats, we consider the standard measures, i.e., precision, recall, F-measure, and accuracy [51] to evaluate the effectiveness of a bug fixing patch identifier.

5.7 Chapter Conclusion

In this chapter, we revisit the bug fixing patch identification task (proposed in Chapter 4) with a new Convolutional Neural Networks (CNN) based approach. This

approach takes both the commit log and preprocessed patch code as input and automatically learns representations/features from the patch for better classification.

We recollect a new set of stable (bug fixing patches) and non stable patches. This new dataset contains 7,793 patches. To evaluate the performance of our new CNN-based approach, we perform a 10-fold cross validation on the data set and calculate precision, recall, F-measure for both a baseline approach (i.e., LPU+SVM based proposed in Chapter 4) and the CNN-based approach. Our results show that although the CNN-based approach could not achieve the same recall as the baseline (a relative 17% decrease), it could achieve a higher precision (a relative improvement of 187%), F-measure (a relative improvement of 71%), and accuracy (a relative improvement of 32%). We have also checked the agreement between two approaches, and find that the CNN-based approach and the LPU+SVM based approach have 70% agreement on all testing patches. The low $kappa$ statistic shows the potential of combining the two approaches for a better classifier, which will be our future work.