Automatic Contextual Commit Message Generation: A Two-phase Conversion Approach

Md Salman Ahmed*, Anika Tabassum*
*Department of Computer Science
Virginia Polytechnic Institute and State University
{ahmedms, anikat1}@vt.edu

***Abstract*—Generating high-quality commit messages are very important in maintaining records of changes for bug fixes and feature additions in software. Short commit messages can contain vague information about changes while long messages may contain unnecessary information. To the best of our knowledge, all state-of-the-art techniques fail to fulfill the complete requirements of clear and contextual commit messages. Thus, this paper discusses the conversion of *diffs* using a two-phase conversion approach (*diffs* to natural language, natural language to commit message) to fulfill the requirements of clear, contextual, and precise commit messages. The first phase of the approach describes source code changes in human understandable natural language. The second phase adopts the recent neural machine translation technique to produce contextual commit messages. We use recent quantitative score like Bleu to evaluate the quality of the commit messages produced by the approach.**

I. Introduction

Change is a big part of the software development and maintenance. A commit message describes the change between two code versions in a software. This description includes but not limited to development tasks, bug fixes, new feature additions, and device supports during a software's life cycle. Commit messages are crucial for large and complex software. This kind of software often changes due to the discovery of new bugs or the addition of new features. However, an addition of these new features or bug fixes can have side effects to the overall software systems. This is why developers utilize commit messages for documenting their changes to trace back the cause of a side effect. In addition, a large-scale software usually requires multiple developers in its life cycle. Commit messages play a crucial role by providing the contexts of other developers' changes. A good commit message describes a software change clearly and concisely. Usually, a commit message contains a short (50 characters or less) summary or title followed by a contextual description [2]. However, too short commit messages may convey vague information, whereas long commit messages may convey unnecessary information.

The purpose of commit messages is to provide clear, contextual, and precise information for developers to review and understand code changes. Unfortunately, many developers neglect writing meaningful commit messages or write vacuous commit messages such "initial commit", "bug fix", "more work", "minor changes", "oopsie", etc. Though commit messages are meant to be helpful, this type of commit messages does not provide any information. To address this issue, researchers have been working actively to assist developers writing commit messages by automatically generating the commits. Some researchers have been working to correlate the **diffs** with commit messages. To do so, they try to utilize the deep learning-based approaches. For example, Jiang et al. propose a neural machine translation (NMT) technique to correlate **diffs** to commit messages [14]. However, their technique generates an equal number of bad commits as good commits. As a result, they require human assessments to refine their results. Another tool Code-NN [13] also uses the NMT technique, but the tool summarizes code snippets. Thus, the need for clear, contextual, and precise commit messages, especially the title summaries is still demanding.

In this paper, we present an automatic commit message generation technique using a two-phase **diff** conversion approach. The first phase of the two-phase conversion approach converts the difference between two revisions of a project into readable natural language sentence fragments. In this phase, we utilize GumTree [10], the state-of-the-art fine-grained code differencing tool to extract the changes between two versions of a source file. By combining the changes of all source files in a project, we get the difference between two versions of a project. In addition to the change extraction in this phase, we describe the changes using natural language sentence fragments[1]. In the second phase, we train NMT models to translate the generated sentence fragments to the original commit messages. To evaluate the effectiveness and quality of the translations, we utilize the top starred and actively maintaining Java projects in GitHub.

This research makes three key contributions:

1) We convert the changes between two versions of a project into natural language sentence fragments.
2) We map the generate natural language-based change description to the original commit messages by training two neural machine translation models.
3) We evaluate the effectiveness and quality of the neural machine translation models by utilizing top starred and actively maintaining Java projects in GitHub.

[1]A sentence fragment does not express a complete thought because it does not have subjects.

II. Overview

The two-phase commit message generation technique utilizes the state-of-the-art tools and techniques. Figure 1 illustrates the overall architecture of the two-phase commit message generation system. The top and bottom rows of the diagram correspond to the first and second phases of the system respectively.

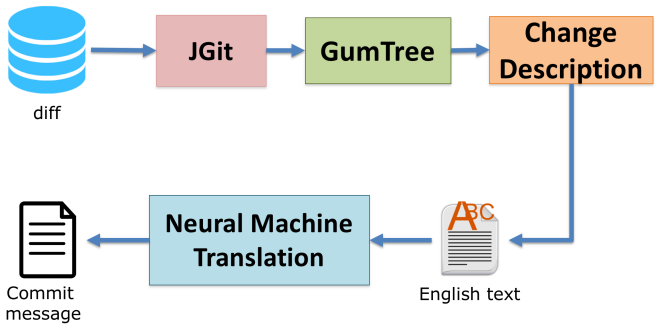


Fig. 1. Overall architecture of the two-phase commit message generation

A. Phase 1

In the first phase, we utilize the JGit extension[2] to extract diffs and associated commit messages. We also extract the file contents, the number of modified files, and commit date by each revision. We then pass the modified files through GumTree [10] and extract fine-grained differences between two phases. Though the latest version of GumTree has support for extracting code differences in many programming languages, we restrict our work for only Java. By default, GumTree has the support for edit script generation and web-based change visualization. Since we are interested in both fine- and coarse-gained changes, GumTree's default edit script or web-based diff does not quite serve our purpose. So, we add new functionalities in the GumTree implementation to support both the fine- and coarse-grained change extraction. After extracting the code changes, we pass the changes to a translation unit which translates each change to a phrase[3] (or a group of words). A natural language generation unit then converts the generated phases to English sentence fragments.

B. Phase 2

In the second phase, we pass the generated sentence fragments to an NMT unit to generate commit messages so that our final commit messages reflect the context and developers' intents. To do that, we extract the top starred and actively maintained Java project such as elasticsearch [1], hadoop [4], spring-framework [8], spring-boot [7], mockito [5], guava [3], and slf4j [6]. A short description of each of seven projects is available in the appendix A. We describe all revisions of each project and generate necessary datasets and vocabularies for the NMT models. To compare the effectiveness and performance of NMT models, we train the long short-term memory based sequence to sequence and attention NMT models. We provide a basic overview of the two NMT models in the appendix B.

[2]http://www.eclipse.org/jgit/

[3]Please note that English grammar may treat a group of words a clause, but here the group of words does not have subject or predicate. That is why we considered the group of words as a phrase to be on the safe side

III. Approach

The two-phase commit message generation technique generates a commit message by analyzing all the added, modified, and deleted files from the current head to the latest stage of a Git repository. First, the technique describes the changes due to the addition, modification, and deletion of source files into English sentence fragments. Then, an already pretrained NMT model translate the generated sentence fragments to a commit message. Thus, the technique performs several steps to generate sentence fragments and train NMT models. We describe the steps in the following subsections.

A. Change Extraction

In this step, we extract the changes between two revisions of a project from a Git repository. JGit extension, a lightweight pure Java library, has all the required functionalities to control the Git version control system programmatically. We use JGit APIs to retrieve all the added, modified, and deleted files as well as the contents of each modified file before and after modifications in a revision (or commit). In each revision, we retrieve the modified files and contents by comparing the revision with the previous revision. We ignore the first revision because there is no previous revision to compare.

Next, we pass the extracted files from a revision to GumTree for extracting the source code changes. Since GumTree takes a source version and a destination version of each file, we pass an extracted file before modification as the source file and after modification as the destination file. In case of added and deleted files, an empty file replaces a source file for an added file and an empty file replaces a destination file for a deleted file. We pass each added, modified, and deleted file to the GumTree iteratively to extract the changes. At this stage, astute readers must realize that a large number of extracted files from a revision can reduce the overall performance of the change extraction process. A large number of extracted files also produce huge change description which in turn affect negatively in the training process of the NMT models. That is why we calculate the average number of files changed over time in a project repository and consider those revisions that have the number of files changed less than or equal to the average number. It is also important to note that sometimes a revision contains a huge number of changed files due to the renaming of a project repository or merging from a different branch. We ignore this type of revisions (or commits) before calculating the average number of changed files over time. Figure 5 and 3 illustrate the number of files changed over time in spring-framework and elasticsearch project. Table I illustrates the average number of files change over a project's timeline for each of the seven GitHub projects. Both of the

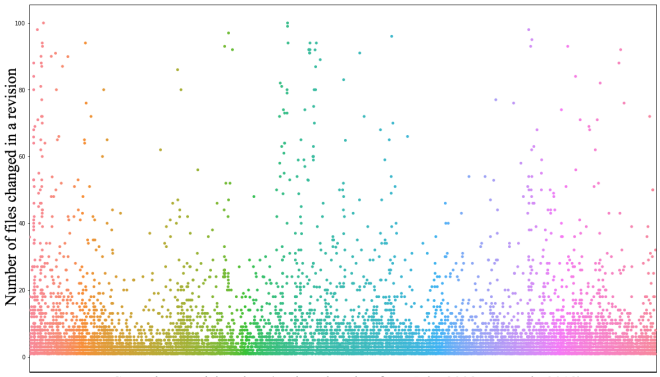


Fig. 2. Number of files changed in each revision of spring-framework project from July 2008 to March 2018

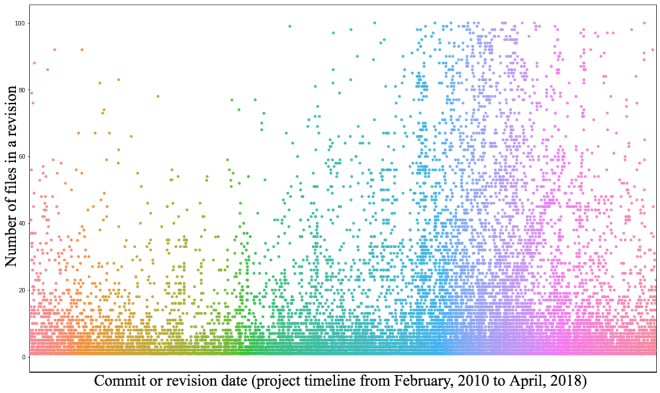


Fig. 3. Number of files changed in each revision of elasticsearch project from February 2010 to April 2018

TABLE I
AVERAGE NUMBER OF FILES CHANGED OVER A PROJECT TIMELINE

Project name	Average number of files changed in each revision
hadoop	5
slf4j	5
elasticsearch	9
spring-boot	5
spring-framework	4
mockito	5
guava	4

figures and the table confirm that the average number of the file changed in a revision is 5. However, we see that the average number of the file changed for elasticsearch is 9. That is why we conservatively decide to pass a total of 10 changed files to GumTree to extract the source code changes.

By default, GumTree supports five important source code change types such as modification (SRC UPD) and deletion (SRC DEL) of contents from a source file, addition (DST ADD) and deletion (DST DEL) of contents from a destination file, moving (SRC MOV) contents of a source file from one location to another, and moving (DST MOV) contents of a destination file. However, we assume that the SRC MOV and DST MOV change types are not significant for a commit message generation for simplicity. The SRC UPD change type does not fit in the context of commit message generation because we are more interested in the updates of destination file than the source file. Thus, we consider only SRC DEL, DST ADD, and DST UPD change types.

```
 * @see ReflectionUtils
@@ -51,8 +51,11 @@ public abstract class ClassUtils {
        /** Suffix for array class names: "[]" */
        public static final String ARRAY_SUFFIX = "[]";

-       /** Prefix for internal array class names: "[L" */
-       private static final String INTERNAL_ARRAY_PREFIX = "[L";
+       /** Prefix for internal array class names: "[" */
+       private static final String INTERNAL_ARRAY_PREFIX = "[";
+
+       /** Prefix for internal non-primitive array class names: "[L" */
+       private static final String NON_PRIMITIVE_ARRAY_PREFIX = "[L";

        /** The package separator character '.' */
        private static final char PACKAGE_SEPARATOR = '.';
@@ -85,6 +88,12 @@ public abstract class ClassUtils {
         */
        private static final Map<String, Class> primitiveTypeNameMap = new HashMap<String, Class>(16);

+       /**
+        * Map with common "java.lang" class name as key and corresponding Class as value.
+        * Primarily for efficient deserialization of remote invocations.
+        */
+       private static final Map<String, Class> commonClassCache = new HashMap<String, Class>(32);
+

        static {
                primitiveWrapperTypeMap.put(Boolean.class, boolean.class);
@@ -98,6 +107,7 @@ public abstract class ClassUtils {

                for (Map.Entry<Class, Class> entry : primitiveWrapperTypeMap.entrySet()) {
                        primitiveTypeToWrapperMap.put(entry.getValue(), entry.getKey());
+                       registerCommonClasses(entry.getKey());
                }

                Set<Class> primitiveTypes = new HashSet<Class>(16);
@@ -108,9 +118,25 @@ public abstract class ClassUtils {
                for (Class primitiveType : primitiveTypes) {
                        primitiveTypeNameMap.put(primitiveType.getName(), primitiveType);
                }
+
+               registerCommonClasses(Boolean[].class, Byte[].class, Character[].class, Double[].class,
+                               Float[].class, Integer[].class, Long[].class, Short[].class);
+               registerCommonClasses(Number.class, Number[].class, String.class, String[].class,
+                               Object.class, Object[].class, Class.class, Class[].class);
+               registerCommonClasses(Throwable.class, Exception.class, RuntimeException.class,
+                               Error.class, StackTraceElement.class, StackTraceElement[].class);
        }

+       /**
+        * Register the given common classes with the ClassUtils cache.
+        */
+       private static void registerCommonClasses(Class... commonClasses) {
+               for (Class clazz : commonClasses) {
+                       commonClassCache.put(clazz.getName(), clazz);
+               }
+       }
+
        /**
         * Return the default ClassLoader to use: typically the thread context
         * ClassLoader, if available; the ClassLoader that loaded the ClassUtils
@@ -188,6 +214,9 @@ public abstract class ClassUtils {
                Assert.notNull(name, "Name must not be null");

                Class clazz = resolvePrimitiveClassName(name);
+               if (clazz == null) {
+                       clazz = commonClassCache.get(name);
+               }
                if (clazz != null) {
                        return clazz;
                }
@@ -200,16 +229,16 @@ public abstract class ClassUtils {
                }

                // "[Ljava.lang.String;" style arrays
-               int internalArrayMarker = name.indexOf(INTERNAL_ARRAY_PREFIX);
-               if (internalArrayMarker != -1 && name.endsWith(";")) {
-                       String elementClassName = null;
```

Fig. 4. A sample diff of a revision from spring-framework

We observe that a revision may contain a minor or a big change in a file. For example, we can see in Figure 4 that parts of a revision in spring-framework contain the addition and modification of class fields, the addition of a method, and deletion of a code block. Blueish color represents the additions, yellow color modifications, and reddish color deletions. The addition of a method is a big change because GumTree generates a lot of edit scripts for the addition. However, the addition or modification of the class field is a minor change because GumTree generates a few scripts for that change. Since we are interested in generating natural

language description of the changes, and subsequently commit message from the natural language description, fine-grained change (e.g., method addition) could produce a very large change description. Consequently, the large description may have negative impacts on the NMT models. That is why we add functionalities in GumTree to support for coarse-grained change extraction as well as the default fine-grained change extraction. GumTree produces two context trees (source context tree and destination context tree) after analyzing the differences between a source file and a destination file. We recursively traverse the two trees and limit the depth of a change where necessary. More specifically, we stop finding fine-grained changes as soon as we get a high-level change. For example, a code block in a method contains statement changes, variables' value changes, other method invocations, etc. We don't track all of these changes, rather we describe these changes as a block change. In Figure 4, we extract the fine-grained changes for class field additions or modifications and coarse-grained changes for the method addition. It is also important to note that the scope of this work is to extract only the class, field (or attribute), and method level changes. Javadoc, import or package declarations are out of the scope of this work.

B. Code Change Description

In this step, we categorize each change type by separating class, attribute, and method level changes. A tuple of four elements represents each change. In the recursive traversal process of GumTree, we track the class, method, and attribute level changes and generate a tuple for each change. List 1 shows the tuples of changes highlighted in Figure 4. Lines 1 to 11 represent the additions, line 13 modification, and lines 15 to 17 deletions. An empty 'Field' with non-empty 'Method' in the tuple represents a method level change, and vice-versa. Empty 'Field' and 'Method' in the tuple represent a class level change.

```
1  /*----------------------------------------------------------------------*/
2  Type: IfStatement , Field: , Method: forName, Class: ClassUtils
3  Type: FieldDeclaration , Field: commonClassCache, Method: , Class: ClassUtils
4  Type: IfStatement , Field: , Method: forName, Class: ClassUtils
5  Type: ExpressionStatement , Field: , Method: , Class: ClassUtils
6  Type: ExpressionStatement , Field: , Method: , Class: ClassUtils
7  Type: MethodDeclaration, Field: , Method: registerCommonClasses, Class: ClassUtils
8  Type: IfStatement , Field: , Method: forName, Class: ClassUtils
9  Type: ExpressionStatement , Field: , Method: , Class: ClassUtils
10 Type: FieldDeclaration , Field: NON_PRIMITIVE_ARRAY_PREFIX, Method: , Class: ClassUtils
11 Type: ExpressionStatement , Field: , Method: , Class: ClassUtils
12 /*----------------------------------------------------------------------*/
13 Type: StringLiteral : "[", Field: INTERNAL_ARRAY_PREFIX, Method: , Class: ClassUtils
14 /*----------------------------------------------------------------------*/
15 Type: IfStatement , Field: , Method: forName, Class: ClassUtils
16 Type: VariableDeclarationStatement , Field: , Method: forName, Class: ClassUtils
17 /*----------------------------------------------------------------------*/
```

Listing 1. Tuples of changes. A tuple consists of four elements (type, attribute, method, and class name)

C. Natural Language Generation

We convert the tuples of changes into human-readable sentence fragments in this step. A natural language generation unit groups the tuples by an attribute and a class or by a method and a class. The natural language generation unit then converts each group into a readable sentence fragment. In this step, we also utilize a translation unit that translates the 'Type' element of a tuple to a human-readable format. For example, the translation unit converts 'ExpressionStatement' to 'statement', 'SimpleType' to 'type', 'ArrayType' to 'array', etc. GumTree identifies more than 30 change types. We maintain a config file that contains all of the translations. It is important to mention that the config file is tunable for good translations. List 2 illustrates the natural language description of the highlighted part of Figure 4.

```
1 updated registerCommonClasses method in ClassUtils by adding
  method declaration . updated forName method in ClassUtils by
  adding if statement . added attributescommonClassCache, and
  NON_PRIMITIVE_ARRAY_PREFIX. updated attribute
  INTERNAL_ARRAY_PREFIX. updated forName method in
  ClassUtils by removing if statement, and variable declaration
  statement.
```

Listing 2. Natural language description of the highlighted change in Figure 4

D. Commit Message Generation

The end goal of this work is to suggest developers a commit message generated automatically by analyzing all the added, modified, and deleted files from the current head to the latest stage of a repository. As described above, we pass all the added, modified and deleted files to GumTree and an NLG unit to get the change description. In this step, a NMT unit translates the change description to a high-level commit message. However, the high-level commit message should be contextual and should reflect the developers' intents. That is why we first train the NMT models using all existing revisions (or commits) so that the model can learn developers' commit styles. We assume that a project has already many revision logs. Consequently, the automatic commit message generation in the context of a newly kickstarted project is out of the scope of this work.

IV. Implementation

A. Change Extraction and Description

As mentioned earlier, we use JGit for extracting modified files from a project's repository, GumTree (modified) for extracting the fine- and coarse-grained changes, and our own implementation for describing the changes. We implement a Java-based prototype of 2k lines of code. We add 430 lines of code in GumTree to add recursive traversals for the fine- and coarse-grained changes.

B. Commit Message Generation

We write a few python scripts for data preprocessing and dataset generation for NMT models. However, we use a readily available NMT package [15] for training. The prototype of our implementation along with the python scripts are available on GitHub[4].

1) Dataset for NMT: To collect the train and test datasets for the NMT models, we repeatedly generate change descriptions for all revisions in a project by following the approach described in section III. We store the generated change description along with the original commit message, revision number, and the commit date.

[4]https://github.com/salmanyam/CommitGen

2) Preprocessing for NMT: Before starting the preprocessing, we collect the total size of all diffs in each of the seven projects' repository. Table II shows the total diff sizes for all projects. Analyzing the diff dataset for each project gives us interesting statistics about the dataset. We determine the median and mean token counts for each project by tokenizing all diffs in a project. Figure 5 and 6 show the box plots for spring-framework and hadoop, respectively. The dotted line inside a box represents the mean value. Table II also shows the median and mean token counts for each project. Since the total diff size for elasticsearch is extremely high (167 Gigabytes), we are unable to calculate the statistics for elasticsearch. As the table and boxplots illustrate, the median number of tokens for each project varies from 200 to 500 and mean from 600 to 1000. That is why we conservatively discard the diffs with more than 1000 tokens for NMT training. It is also important to note that we also discard the diffs for merge commits and diffs with more than ten Java source file modifications (as the reason discussed in section III-A)

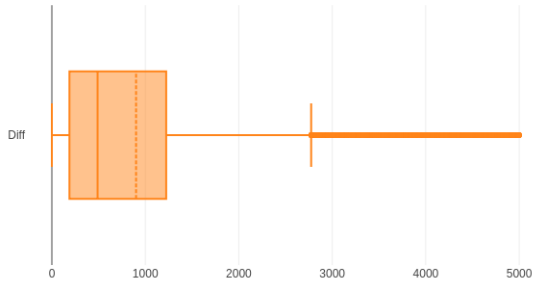


Fig. 5. Boxplot using the token count in spring-framework project

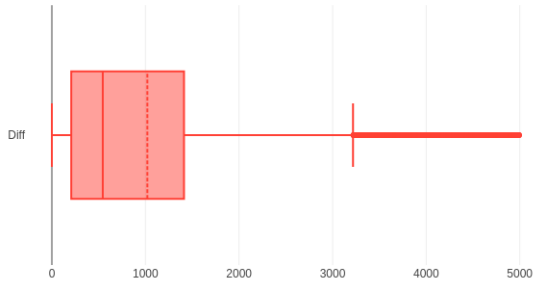


Fig. 6. Boxplot using the token count in hadoop project

3) Neural Machine Translation: In this part of the work, we train two NMT models as mentioned in the Overview section. The first model is a sequence to sequence NMT model whereas the second model is an attention-based NMT model. Both models use a a long short-term recurrent neural network. The purpose of using two models is to show the comparative performances.

- **Vocabulary**: A neural machine translation model in our case requires a set of vocabulary dictionaries for the original commit messages and the generated change descriptions. We tokenize the original commit message and change description based on space to build the two dictionaries. Since the class, attribute, or method names are important parts of the vocabulary, we do not change the case of the dictionary words. Though this could increase the total size of the vocabulary, it is important for a quality translation.
- **NMT dataset**: For the training data, we randomly select 80% rows from a dataset after preprocessing. We use the rest of the data for the test and validation. We split the rest 20% data equally for the test and validation dataset.
- **NMT models**: Implementation of the two models is available on GitHub[5]. We train each of two models with different configurations. For each of the models, we use a 2-layer LSTM, 128 hidden dimensions, 12k training steps, and 0.2 dropout value. Table III shows the configurations and evaluation metrics for different projects.

V. Evaluation

In this section, we evaluate the practicality of the approach by generating commit messages using the GitHub projects and matching with the corresponding original commit messages. To check the similarity between the generated commit messages and original commit messages, we use the BLEU (bilingual evaluation understudy) score. BLEU score is the most widely used method for automatically quantifying the quality of machine translation. BLEU score is usually determined by the score of 0-100% or 0-1. More details regarding the BLEU score is available in the appendix C. We perform the evaluation on Ubuntu 64-bit operating system with Intel Xeon(R) CPU E5-1620 v2 @ 3.70GHz 8 processor and 16 Gigabytes of memory.

A. Generated Commit Message Analysis

In this section, we describe some of the generated commit message translated by the neural machine translation models. List 3 and 4 show some of the good translations and bad translations of the sequence-to-sequence and attention-based model, respectively. In the lists, 'src' means the change description, 'ref' means the original commit message, and 'nmt' means the translated commit message. For the sake space, the 'src' is truncated into a line, but originally the 'src' is several line descriptions. It is also important to note that the quality of most of the translations is bad. The term 'bad' means that there is not much word to word similarity. However, the translated commit messages may be semantically similar with the original commit messages. This analysis is the out of the scope of this work.

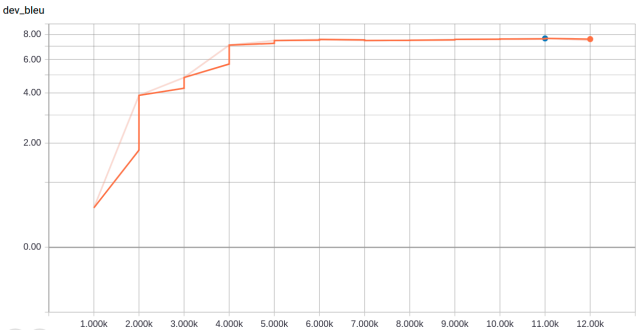
[5]https://github.com/tensorflow/nmt

TABLE II
DIFF STATISTICS OF EACH PROJECTS ON GITHUB

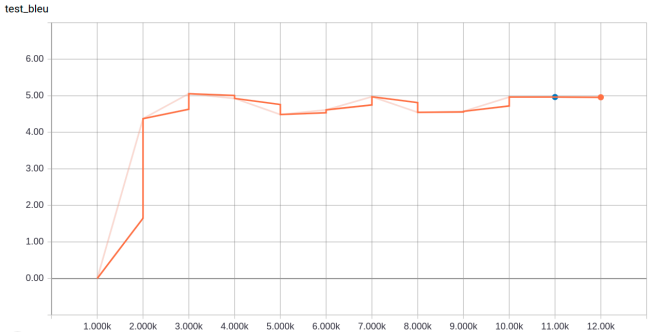
Project Name	Diff size	Median number of tokens in tokenized diffs	Avg. number of tokens in tokenized diffs	Total revisions	Number of revisions having a diff of less or equal to 500 tokens	Number of revisions having a diff of less or equal to 1000 tokens
elasticsearch	167 GB	N/A	N/A	38745	N/A	N/A
hadoop	7.6 GB	544	1020	19231	4175	6451
spring-framework	20 GB	489	900	16460	3606	5815
guava	132.3 MB	454	930	4689	1220	1750
mockito	84.6 MB	253	654	4660	748	1191
spring-boot	29.1 GB	183	587	16370	2233	3374
slf4j	19.5 MB	280	697	1567	205	286

TABLE III
CONFIGURATIONS AND EVALUATION METRICS

Project name	Configuration	Common configuration	BLEU score	
			Sequenceto sequence model	Attention-basedmodel
spring-framework	Training data = 2884 Dev data = 361 Test data = 361 Vocabulary of change description = 41k Vocabulary of original commit = 12k	Layer = 2 Hidden dimension = 128 Training step = 12k Dropout = 0.2	best bleu at step: 11000 dev bleu: 0 test bleu: 0.4	best bleu at step: 5000 dev bleu: 0.3 test bleu: 0.4
hadoop	Training data = 3340 Dev data = 417 Test data = 418 Vocabulary of change description = 47k Vocabulary of original commits = 29k		best bleu at step: 12000 dev bleu: 2.1 test bleu: 1.3	best bleu at step: 11000 dev bleu: 2.4 test bleu: 1.5
spring-boot	Training data = 1786 Dev data = 223 Test data = 224 Vocabulary of change description = 19k Vocabulary of original commit = 6k		best bleu at step: 9000 dev bleu: 2.2 test bleu: 2.5	best bleu at step: 3000 dev bleu: 2.6 test bleu: 2.9
baseline	Training data = 26k Dev data = 3k Test data = 3k Vocabulary of diffs = 50k Vocabulary of original commits = 17k		best bleu at step: 9000 dev bleu: 3.7 test bleu: 3.4	best bleu at step: 12000 dev bleu: 3.5 test bleu: 3.5



(a) BLEU score of the validation dataset in different iterations



(b) BLEU score of the test dataset in different iterations

Fig. 7. BLEU scores for the experiment with smaller diffs

```
src: added inactive time ns in check idle method of index shard class ...
ref: add test only api that allows to pass the inactive time in ns directly if we set the inactive time for the shard via api the entire test if fully time dependent and might fail if we concurrently check if the shard is inactive while the document we are indexing is in flight
nmt: use assert that fail read latest lucene

src: updated MockMvcClientHttpRequestFactory method ...
ref: Polishing cherry picked from commit aaded
nmt: removed redundant <unk> code code <unk> the new box implementation

src: updated initBinder method in ServletAnnotationControllerHandlerMethodTests ...
ref: Fix failing test
nmt: Avoid unit tests since they for Spring MVC
```

Listing 3. Good translations using both nmt models

```
src: updated getTrashCanLocation method in ViewFileSystem by removing method declaration.
ref: Remove stale method ViewFileSystem#getTrashCanLocation.
nmt: Remove duplicated jar <unk> latest 4.0.x.Final.

src: updated onContainerResourceUpdated method in ServiceScheduler by adding method declaration. updated onUpdateContainerResourceError method in ServiceScheduler by adding method declaration.
ref: Rebased onto trunk — fix conflicts
nmt: MAPREDUCE-4414. Add main methods <unk> JobConf and YarnConfiguration, for debug purposes.

src: updated loadImage method in ImageLoaderCurrent by adding method invocation.
ref: OfflineImageViewer incorrectly passes value of imageVersion when visiting IS_COMPRESSED element.
nmt: Added toString for failed <unk> SequenceFileAsBinaryOutputFormat.WritableValueBytes <unk> SequenceFileAsBinaryOutputFormat.WritableValueBytes <unk> re-introducing missing constructors.
```

Listing 4. Sample bad translations using both models

B. Baseline Analysis

We analyze the baseline work [14] by reproducing its models from the available dataset. One important thing is to note that the baseline considers the diffs with maximum 100 tokens. List 5 shows an example of the baselines diff. This is not quite realistic as we show that the GitHub projects have average 200-500 tokens per diff. The baseline models use nematus [20] as its neural machine translation model. We fail to reproduce the baseline models using the numatus machine translation model as nematus requires the GPU computation. Without a GPU support, the training process is 100x slower with only the CPU. Table III shows the results for the baseline as well. List 6 shows some of the translations generated by the sequence to sequence model using the baseline data. As the list illustrates, some of the translations are good. That is why we perform an experiment with smaller diff as discussed in the next section.

```
mmm a / src / gwt / src / org / rstudio / core / client / widget / ModalDialogBase . java <nl> ppp b / src / gwt / src / org / rstudio / core / client / widget / ModalDialogBase . java <nl> public abstract class ModalDialogBase extends DialogBox <nl> protected void onDialogShown ( ) <nl> { <nl> + if ( okButton_ ! = null ) <nl> + FocusHelper . setFocusDeferred ( okButton_ ) ; <nl> } <nl> protected void addOkButton ( ThemedButton okButton ) <nl>
```

Listing 5. Sample diff in baseline's dataset

```
ref: Added back to action bar ( no Blog )
nmt: always focus ok button in message dialog

ref: excluded produced artifact from the dependencies so not included in the ' optional ' dir
nmt: Added xml — smali to the " starter "

ref: Prepare next development version .
nmt: prepare for next development iteration

ref: use read write optional writable
nmt: remove unnecessary call <unk> transport layer transport shared operation action
```

Listing 6. Translations produced by the baseline model with the baseline's dataset

C. Experiment with Smaller Diffs

In this experiment, we split a diff into smaller chunks and make several diffs from a single diff to evaluate the performance of the neural machine translation models. We use a reduced configuration (train data = 875, validation data = 109, test data = 110, vocabulary of change description = 47k, and vocabulary of original commit = 29k) for this experiment. In this experiment, we see the rise of BLEU scores for both validation (7.6) and test (5.0) dataset. Figure 7(a) and 7(b) illustrate the tensor board output for the validation and test BLEU scores.

D. Runtime Performance Evaluation

We also evaluate the runtime performance of the human-readable change description from the source code differences. To evaluate the technique accurately, we isolate the time required for GumTree to generate the differences between two source codes. Figure 8 illustrates the runtime time performance of our approach as well as the GumTree. As illustrated by the figure, GumTree's performance remains constant over the life-cycle of a project, i.e., GumTree's performance does not vary too much with source content. However, our approach creates some temporary files in order to pass those to GumTree. For example, a Git repository only contains the change between two versions, that is why we create two complete files for two revisions and store these files as temporary files. The time of these disk operations increase as the contents of a file increase. Thus, we see a stepping slope for the total time.

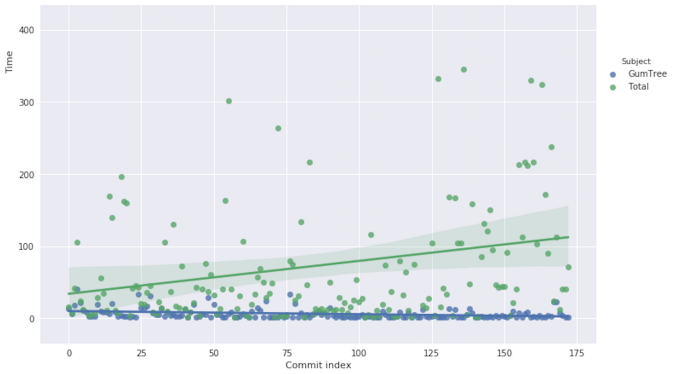


Fig. 8. Runtime overhead increases as the project life-cycle moves forward, GumTree's performance remains constant

VI. Thread to Validity

Though there is no human evaluation this work, there is no human biases. However, we still make some assumptions regarding the change description, and data preprocessing. We extract coarse-grained changes in many places which may not be accurate. Conversely, fine-grained changes may not be applicable for the commit message generation. A rigorous research in this area can confirm the applicability of fine-grained and coarse-grained changes in the context of high-level commit message generation. We also discard those revisions that contain more 10 modified files assuming that the more number of the file changed in a revision the bigger the size of the diff is. However, this may not be always true in reality.

VII. Related Work

The primary goal of this paper is to generate clear and contextual commit messages. We try to achieve this goal by improving the techniques discussed in [14]. In [14], Jiang et.

al. propose a model for generating short commit messages by directly translating **diffs**. They use a recurrent neural network (RNN) attention model. However, their model fails to produce high quality commit messages for larger **diffs** as they directly translate **diffs** to short texts. We improve their techniques by describing **diffs** to natural language and then natural language to commit messages. For this two-phase approach (**diffs** to natural language and natural language to commit messages), we utilize many state-of-the-art techniques including the fine-grained source code differencing tool GumTree.

Several state-of-the-art techniques [11], [13], [16], [21] summarize Java source codes. These techniques analyze the control flow of methods in a source code and summarize the methods. Most of the authors in these papers use Software usage model (SWUM) to tokenize the keywords from Java methods, identify important statements from code and generate text using the extracted code statements. In [17], Nazar et. al. also discuss the recent advancements, tools, libraries, and evaluation processes used in source code summarization. For example, the authors use vector space model and latent semantic indexing for summarization in [11]. In [13], the authors propose a model named CodeNN where they use LSTM neural attention model to summarize C# and MySQL source codes to texts.

Natural language generation (NLG) technique helps to produce understandable text in English or other human languages [18]. Researchers in software engineering also use the technique to describe source codes using understandable text in English [9]. In [12], Hill et. al. suggests how to extract contextual information from source codes by explaining identifiers and word usages. Buse et. al. propose an architecture named 'Deltadoc' for generating texts from program changes [9]. To generate text, they describe the path predicates of every statement of a source code. Then they apply NLG to fix the dependency and flow of the generated text.

Researchers improve text summarization greatly using a pointer generator network in machine translation. The authors in [19] uses a hybrid pointer-generator coverage network. The hybrid pointer-generator coverage network copies words from the source text via pointing and helps accurate reproduction of information. The coverage part of the technique keeps track of what has been summarized. This hybrid pointer-generator model handles out-of-vocabulary words when summarizing a text. This model also avoids repetition of same texts.

VIII. Discussion

We face many challenges for generating the change description for the GitHub projects and training the neural machine translation models. Though the change description generation is less rigorous, it takes a significant amount of time to generate all the change descriptions for a project. All the open-source neural machine translation models need GPU support for efficient computation. Running those models without GPU takes 100x more time. Since our work environment does not have GPU support, we use CPU for all the training and testing. That is why we could not be able to use the nematus machine translation package which is more optimized than the one we use in this work.

The original commit messages contain many bad data including bug ID, project name, contributor name, special characters, and Unicode characters. Different projects contain these bad data in commit messages differently. We are able to clean the special characters, Unicode characters, and some bug IDs. So, we will take a more generic approach to clean up these bad data from commit messages in future. For simplicity, we ignored Javadocs while generating the change description. However, we realize that Javadoc may contain important keywords for a contextual commit message. In future, we will incorporate Javadocs in our change description generation process.

For Neural machine translation we first planned to use state-of-the art automatic summarizer deep learning models like sequence to sequence and pointer generator network model [14]. For summarization, since the natural language also contains a lot of Java based words the summarizer was not producing any correct results. Because, pointer generator when trained on dataset generates a vocabulary as each token as a word, so it was considering every method name, class name in natural language as a new vocab which it was later unable to find in test dataset. Also, after generating vocabulary it fixes weights to each vocab which to point out as the summarizer does not have any reference target output text to generate it points out words only from the reference input text, i.e., natural language. As a result, When we trained this summarizer on our natural language it was producing a very short meaningless message. Figure 9 is showing some results pointer generator was producing after summarization. Due to this we planned to use machine text translation models instead of summarization models.

As part of using neural machine translation models, we planned to replicate the model built by [14] as our idea was based on that paper. However, their model was built over Tensorflow GPU and we were unable to provide GPU and use their model. So, we considered two popular and state-of-art neural machine translation models- sequence to sequence and attention encoder decoder model due to availability of its implementation and code over GitHub.

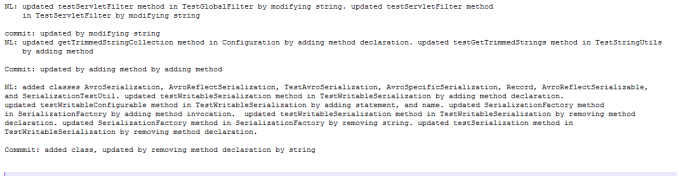


Fig. 9. Commit messages produced due to pointer generator summarizer model

IX. Conclusion

Change description has been recently introduced as a method of generating additional contextual description in a commit with a high-level commit message. However, a very

few researchers tackle the problem of generating a high-level commit message from change description. In this paper, we introduce the idea of two-phase commit message generation approach by utilizing the state-of-the-art source code differencing tool and neural machine translation models. Specifically, we describe the generation of a change description from the two version of a Java source file and contextual commit message from the change description using neural machine translation models. We train the translation models using the existing data from a project's Git repository. We realize that most of the times our change description is fine-grained which is good for describing changes but impacts badly on the neural machine translation models.

In closing, we observe that developers do not follow the standard while writing their commit messages. As we described, many diffs are too large and many commit messages contain bad data. In addition, a contextual and high-level commit indicates the overall purpose of a change. Since it is difficult (even for humans) to write a high-level and contextual short description from a huge change description, neural machine translation based solution may not be a viable solution for the automatic generation of commit messages for big diffs. However, if a repository contains small diffs with a contextual description, it easy is to describe the changes as well as to train neural machine translation models for the automatic generation of future commit messages.

In short, although we failed to produce high Bleu score, which was our only evaluation metric used for machine translation model, our model was able to produce some quality commit messages better than the referenced commit messages.

X. Workload

Our project source code is available on https://github.com/salmanyam/CommitGen. The readme file on our project has all the necessary information regarding the usage of the project.

A. Md Salman Ahmed

He designed and implemented the JGit wrapper functionalities, natural language generation unit, translation unit, and train data generation functionalities in the commit-gen project. He also added the support for fine- and coarse-grained change extraction mechanisms in GumTree using recursive traversal in GumTree's source and destination context trees. He prepared datasets for neural machine translation models and trained two neural machine translation models. As a part of the write-up, he also wrote all the parts associated with his implementations and the evaluation. He also setup all GitHub repositories related to this work.

B. Anika Tabassum

She did the statistical and performance evaluation graphs and tested datasets on different neural machine translation models. She also trained natural language data over pointer generator network model, the most successful automatic summarizer done by Stanford NLP group [6] and tried to replicate the NMT model built by [14] but failed to produce desired results [discussed in Section VIII]. As part of the write-up, she is responsible for the writing related to neural machine translation models.

References

[1] Elastic search. https://github.com/elastic/elasticsearch. Retrieved: 2018-05-07.

[2] Git commit good practice. https://wiki.openstack.org/wiki/GitCommitMessages. Retrieved: 2018-02-25.

[3] Guava. https://github.com/google/guava. Retrieved: 2018-05-07.

[4] Hadoop. https://github.com/apache/hadoop. Retrieved: 2018-05-07.

[5] Mockito. https://github.com/mockito/mockito. Retrieved: 2018-05-07.

[6] Slf4j. https://github.com/qos-ch/slf4j. Retrieved: 2018-05-07.

[7] Spring boot. https://github.com/spring-projects/spring-boot. Retrieved: 2018-05-07.

[8] Spring framework. https://github.com/spring-projects/spring-framework. Retrieved: 2018-05-07.

[9] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42. ACM, 2010.

[10] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[11] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 35–44. IEEE, 2010.

[12] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 232–242. IEEE, 2009.

[13] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2073–2083, 2016.

[14] 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*, pages 135–146. IEEE Press, 2017.

[15] M. Luong, E. Brevdo, and R. Zhao. Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*, 2017.

[16] P. W. McBurney and C. McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2016.

[17] N. Nazar, Y. Hu, and H. Jiang. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5):883–909, 2016.

[18] E. Reiter and R. Dale. *Building natural language generation systems*. Cambridge university press, 2000.

[19] A. See, P. J. Liu, and C. D. Manning. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*, 2017.

[20] R. Sennrich, O. Firat, K. Cho, A. Birch, B. Haddow, J. Hitschler, M. Junczys-Dowmunt, S. Läubli, A. V. Miceli Barone, J. Mokry, and M. Nadejde. Nematus: a toolkit for neural machine translation. In *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 65–68, Valencia, Spain, April 2017. Association for Computational Linguistics.

[21] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[6] https://github.com/abisee/pointer-generator

APPENDIX A
OVERVIEW OF GITHUB PROJECTS

A. ElasticSearch[7]

It is a distributed Restful search API to speed up search process for different queries. For each data, it uses BKD trees to index geotags and numeric data and inverts indices to index the text queries to make the search for a query faster.

B. SLF4J[8]

This is a simple logging faade for Java. This provides an interface for different JAVA logging frameworks. It provides users to build the desired logging framework for his software.

C. Hadoop[9]

It is a framework provided by Apache to run large applications and compute complex algorithms fast using distributed systems. It uses MapReduce algorithm to solve complex problems into several fragments and the smaller problems are distributed over machines to solve parallel. Hadoop also provides HDFS file system for distributed computing.

D. Spring Framework[10]

It is a popular framework for developers to build Java-based web applications. It provides an interface to connect a front-end (UI) to a back-end (database).

E. Spring Boot[11]

This is a faster and easily accessible interface to create and run production-grade Spring-based applications. It provides users to use Spring with less trouble.

F. Guava (Google Core Libraries for Java)[12]

It includes several core libraries of Java such as the list, set, multiset, graph packages, hashes, String processing, I/O etc.

G. Mockito[13]

The most popular unit testing framework for Java, which isolates a portion of the system to test by imitating the behavior of its dependencies.

[7]https://www.elastic.co/products/elasticsearch
[8]https://www.slf4j.org/index.html
[9]http://hadoop.apache.org/
[10]https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html#spring-introduction
[11]https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started- first-application
[12]https://github.com/google/guava/wiki
[13]https://github.com/mockito/mockito

APPENDIX B
OVERVIEW OF NEURAL MACHINE TRANSLATION

A. Sequence-to-sequence model

A deep learning implementation using Encoder Decoder in RNN algorithm. The encoder encodes each word of an input sentence into a vector and thus creates a fixed size vector representation for the input sentence. The decoder decodes the sentence vector found from the encoder and decodes it into a different language. So, each encoder and decoder use an LSTM in their hidden layer.

A decoder during its training uses target output words as input, uses a vector from the encoder and computes the target output sentence. Each target output word in the decoder is feed into an LSTM and it ends with an ending tag ¡EOS¿ in a sentence.

For testing phase, the decoder uses its previously generated target word as its input and computes for next word target output.

B. Attention model

The objective of attention model is to focus on some words of a sentence to produce target output. The encoder computes probability or attention weights for every input words corresponding to its target output word. These attention weights are feed into an LSTM hidden layer. LSTM computes the context vector using attention weights and target input words provided by the decoder. Finally, the context vector along with the hidden state computes the final attention vector or target output words. In the testing phase, each attention vector created in previous time step is used as input in the next time step.

APPENDIX C
BILINGUAL EVALUATION (BLEU) SCORE

[14] An evaluation score used popularly for machine translation. Generally compares reference text with predicted text by counting matching n-gram words of reference text and translated text. Scores are calculated by comparing every translated sentence against reference text and then average of Bleu score of all the sentences gives the combined score. Bleu score is between 0 and 1 $[0 - 100\%]$, where 1 indicates the highest similarity score between the reference and translated text. Each word in the translated text is matched against n-gram word of reference text and then the average score is calculated. Bleu score is just a measure of quality of machine translation with human translation. It is reported to find a very high correlation with human judgement evaluation. However, high Bleu score does not always mean that the quality of translation is good. Because, this score measures based on matching n gram words. However, if the word is synonymous Bleu score is unable to consider it as a matching word with reference text. Python NLTK library provides package to compute Bleu score in both sentence level and corpus level.

[14]https://en.wikipedia.org/wiki/BLEU