Git Good

A Deeper Look at Generating Better Commit Messages
CS 182, Spring 2019
tinyurl.com/gitgoodsp19

Aman Dhar
amandhar@berkeley.edu

Jackson Leisure
jleisure@berkeley.edu

Riku Miyao
rikumiyao@berkeley.edu

Matthew Sit
msit@berkeley.edu

I. Introduction

A. Problem Statement

Git is a distributed version control system that allows multiple users to collaborate on software development projects. Once a user has completed some changes to a codebase, they can save those changes to the repository by creating a commit with those changes. Commits store diff information, showing the lines of code that were updated since the last time a commit was made (Fig. 1). Each commit also has an associated message that should provide a descriptive summary of the changes introduced in the commit.

Currently, there are no widespread introductory programs through which users can learn what makes a good commit message; this only comes through experience–using git for an extended period of time and understanding the intended utility of a message. This, unfortunately, makes it easy for new users to develop bad habits and add undescriptive messages to the commits they create while they are still learning (Fig. 2).

Our goal is to create a set of tools that help users create messages which are good descriptions of the changes they are making. This in the moment guidance would allow new users to gain an understanding of what a typical commit message might look like in their current context, as well as assist experienced users in quickly generating relevant messages.

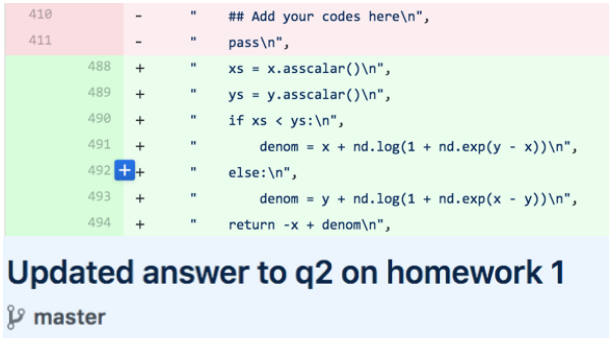

Fig. 1. An example of a diff (upper) with its associated commit message (lower). The red shaded area highlights lines that were removed, while the green shaded area highlights lines that were added. Note that this example is from a Jupyter Notebook written in Python, which was not used in our datasets, since we only used repositories written in Java.

B. Background

Our work is valuable at educational institutions like UC Berkeley since git is generally introduced to computer science students in introductory programming courses, such as Cal's CS 61B: Data Structures. However, it can be difficult for first-time users to know what descriptions are the best when using the git software. For some students, this is not an issue, as they never encounter a situation in which they require the use of the version control component of git. However, many students eventually reach a point at which they must look back at their previous commits, in order to find the exact point at which they made a certain change. If these students do not have adequate commit messages, this process becomes exceedingly difficult and frustrating, both for themselves and for those they are working with. Poor habits formed here can have negative consequences when these students enter industry and other professional settings.

Our goal is to use professional git repositories to train a model which can eventually assist Cal students in determining the goodness of their commit descriptions, in order to allow them to use create more descriptive commit messages.

We hope to achieve this in three ways. First, we hope to create a tool which allows users to see what typical commit messages look like, independent of a specific project or commit message. This should help first time users become acclimated to the typical structure and content of a git commit message, as it can be daunting to decipher what makes a good message when first using git. Second, we want to provide users with a discriminator which assists users by informing them of whether a given commit message is a good fit for the diff that it attempts to describe. This information will provide users with input as to whether their proposed commit message is adequate, as determined by this tool, allowing users to be made aware if their messages are missing key information or are malformed. Third, we intend to give users the ability to use our model to generate complete commit messages, based on a specific commit diff. Providing users with suggested commit messages that are relevant to the changes they have just made would assist new users by displaying realistic examples of messages they should be writing and allow veteran users to spend less time thinking of a descriptive commit message for the code they modified, while preventing both types of users from forgetting key details present in the commit.

With better messages, students will be able to better document their code, allowing them to work more effectively, whether as an individual or as a member of a team.

```
commit 85110ff152de681337c00fc0e5d4f82b2656d87d
Author: Jackson Leisure <jleisure@berkeley.edu>
Date:   Mon Dec 3 22:01:53 2018 -0800

    added notes to ProfileStorage

diff --git a/code/app/src/main/java/com/example/group39/peekmyinterest/EditProfile.java b/code/app/src/main/java/com/example/group39/peekmyinterest/EditProfile.java
index d96243c..f352f85 100644
--- a/code/app/src/main/java/com/example/group39/peekmyinterest/EditProfile.java
+++ b/code/app/src/main/java/com/example/group39/peekmyinterest/EditProfile.java
@@ -139,7 +139,7 @@ public class EditProfile extends ProfileStorage {
             Bitmap bitmap = null;
             try {
                 bitmap = MediaStore.Images.Media.getBitmap(this.getContentResolver(), selectedImage);
-                bitmap = Bitmap.createScaledBitmap(bitmap, 100, 100, true);
+                bitmap = Bitmap.createScaledBitmap(bitmap, 50, 50, true);
                 picbutton.setImageBitmap(bitmap);
                 setPicture(bitmap);
             } catch (FileNotFoundException e) {
diff --git a/code/app/src/main/java/com/example/group39/peekmyinterest/ProfileStorage.java b/code/app/src/main/java/com/example/group39/peekmyinterest/ProfileStorage.java
index 3672eb9..1646896 100644
--- a/code/app/src/main/java/com/example/group39/peekmyinterest/ProfileStorage.java
+++ b/code/app/src/main/java/com/example/group39/peekmyinterest/ProfileStorage.java
@@ -95,6 +95,14 @@ abstract class ProfileStorage extends AppCompatActivity {
         }
     }

+    public void setNotes(String value) {
+        setTierOne("notes", value);
+    }
+
+    public String getNotes() {
+        return getTierOne("notes");
+    }
:
```

```
commit 53fc8b2a390e7b356e65f460a5d71917d5a3746c (origin/justin2, justin2)
Author: Jackson Leisure <jleisure@berkeley.edu>
Date:   Sat Dec 1 18:31:39 2018 -0800

    one edit boi

diff --git a/code/app/src/main/AndroidManifest.xml b/code/app/src/main/AndroidManifest.xml
index eb97440..c75b07e 100644
--- a/code/app/src/main/AndroidManifest.xml
+++ b/code/app/src/main/AndroidManifest.xml
@@ -35,6 +35,7 @@
             android:label="@string/title_activity_personal_profile">

         </activity>
+        <activity android:name=".EditProfile" />
         <activity android:name=".MyProfile" />
         <activity android:name=".ReceivedProfile" />
         <activity android:name=".SentProfile" />
diff --git a/code/app/src/main/java/com/example/group39/peekmyinterest/Connect.java b/code/app/src/main/java/com/example/group39/peekmyinterest/Connect.java
index 3fc9640..96db0b6 100644
--- a/code/app/src/main/java/com/example/group39/peekmyinterest/Connect.java
+++ b/code/app/src/main/java/com/example/group39/peekmyinterest/Connect.java
@@ -148,7 +148,7 @@ public class Connect extends ProfileStorage implements OutcomingNfcManager.NfcAc



-        JSONObject fetchedJson = getMyJson(); //Fetches my json from profileStorage. (real code)
+        JSONObject fetchedJson = getJson(); //Fetches my json from profileStorage. (real code)
         //JSONObject fetchedJson = testJson;

         String fetchedJsonString = jsonToString(fetchedJson);
@@ -210,7 +210,7 @@ public class Connect extends ProfileStorage implements OutcomingNfcManager.NfcAc

             this.tvOutcomingMessage.setText(inMessage); //Displays received Json string in a textview.
:|
```

Fig. 2. Screenshots of example commits with good (upper) and bad (lower) messages associated with the changes in the commit. Note how the good commit message directly relates to the contents of the files which have been altered, while the bad message is vague and unrelated to the changes in the codebase.

II. Related Work

A. Towards Automatic Generation of Short Summaries of Commits [3]

The work in [3] attempts to generate one-sentence commit messages that start with a verb-object pair. First, the authors focus on classifying each diff into one of fifteen separate verb classes. After this, the authors discuss methods of generating messages, though they do not fully explore the commit message generation process. Additionally, they show that it is difficult to achieve high classification accuracy on the verb matching task. This leads us to believe that a classification-based approach that only works on one word at a time may not be an effective way to generate commit messages. Therefore, we do not pursue using similar models as the work in [3], though we do use similar data cleaning and preprocessing techniques. While some of our models generate the commit messages one word at a time, we use recurrent architectures that allow the next generated word to be a function of all previously generated words. We also train a language model assuming the user can specify a good choice for a first word, allowing our models to generate the rest of a commit message with some guidance.

B. Automatically Generating Commit Messages from Diffs using Neural Machine Translation [4]

The work in [4] extends the original work in [3] and attempts to generate commit messages after classifying them according to common sentence structures. This work involves an RNN Encoder-Decoder model to generate messages. We use similar architectures while also experimenting with more complicated, attention-based architectures in an effort to generate more relevant messages. The RNN Encoder-Decoder model's hidden state has a lower dimension than our hidden state sizes, so this should make our model more able to approximate a complex function that maps diffs to commit messages.

III. Approach

A. Data Collection and Processing

We employed the use of two kinds of data, professional repositories and student repositories.

Professional repositories were scraped from GitHub.com, an online service for sharing Git repositories. We made an API call to identify the 100 most starred repositories written in the Java language. Stars on GitHub.com signify repositories that users find interesting, and act as appreciation to the repository owner [6]; it is a measure of community popularity. Due to the high visibility, activity, and community engagement such popular repositories receive, we define their commit messages as the standard for goodness; they are successful in achieving their purpose of informing other users of the history of the codebase. We used a python script to clone each repository, and write the "git log -p" and "git log –format=%s" outputs to .diff and .txt files. We then used those respective files to extract diff information and message information, respectively, for each commit.

60% of these repositories were used to build the training dataset, 20% were used to build the validation dataset, and the remaining 20% were used to build a testing dataset. For each of these three groups, we used only the first 200 commits from each of the associated repositories, due to space limitations. If

a commit message was empty, or contained a default message such as Merge branch or Merge pull request, it was discarded and the next subsequent commit was used as a replacement. No more than 1000 commits were examined in each repository, before moving on to the next. Through this process two dictionaries, one mapping the commit hashcode to the commit message, and one mapping the commit hashcode to the diff content, were created and written as json files, for each of the three dataset splits.

Student repositories were obtained from the Spring 2018 semester of the CS 61B course, here at UC Berkeley. This data is confidential as it contains student work and solutions to actively used programming assignments. The data was used with permission from Professor Joshua Hug. These repositories were processed in the same way as the professional repositories. Instead of dividing these repositories into three groups, a pair of json files, representing the messages and diffs, was generated for each of the 15 selected repositories. There is no need to split the data into groups since this data was not used for training, and a model trained on professional repositories could be used to generate messages on each of these repositories independently.

The 15 student repositories were chosen first by plotting the grade distribution of all of the students in the course (Fig. 3). Three ranges of scores were identified as potential areas of interest: 3200 - 3700 points, 3000 - 3200 points, and 2000 - 2500 points. Five student repositories were randomly sampled from each of these ranges.

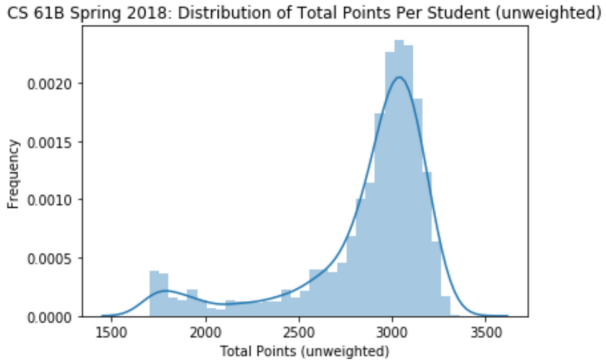

Fig. 3. A distribution plot of the total points students accumulated in Spring 2018. These scores are not weighted by assignment category, as they were in the actual course. This curve can be used to identify repositories from a diversity of experience levels.

B. LSTM-Based Language Model

In our first attempt at commit message generation, we implement and train a language model on commit messages from professional repositories. The language model is an RNN composed of a 1-Layer LSTM with cell size 256. In practice, this model would require at least one starter word in order to begin generating a commit message. While the words generated by the language model may not be specific to a diff, they will be influenced to fill in general language surrounding common changes if the starter word or words are specific to the diff. Additionally, the model generates sequences with <UNK> tokens to represent unknown words that may help complete a more relevant commit message. We discuss how this is handled in the Results section.

C. Transformer-Based Summarization Model

For the transformer model, we tokenized the commit message and diff pairs, and limited the lengths to 30 tokens and 100 tokens, respectively. We next converted them into masked feature vectors by keeping track of a dictionary which kept track of all the words that appeared. These feature vectors were then fed as inputs to our transformer.

As for the architecture, we implemented the attention-based transformer from Vaswani et al. [2] (Fig. 4). We fed this transformer our vectorized inputs with masks and trained it using the Adam optimizer with softmax loss. With large batch sizes (128) the model had training losses around 2.0 but had validation loss of 8.9, which was indicative of overfitting. With smaller batch sizes the model trained slower but did not overfit the data as much, resulting in a validation loss of around 5.0.

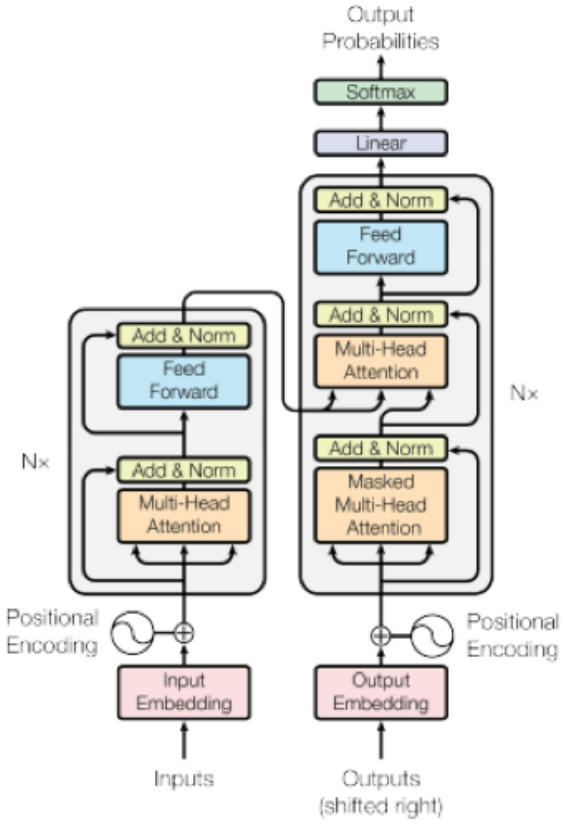

Fig. 4. Diagram of the transformer architecture [2]. We implement this architecture as a component of our summarization model.

D. BERT

We then moved on to the BERT model, a bi-directional transformer (Fig. 5). We used Google's released TensorFlow BASE, uncased (case insensitive) pre-trained model [7], which

performs sentence-pair classification. To fine-tune it to discriminate between good and bad commit messages for a given diff, we fed BERT our professional message and diff pairs, as well as 5 times as many negatively sampled mismatched pairs constructed at random. We performed WordPiece tokenization [8], with a limit of 128 tokens each. Hyperparameter tuning was performed on the learning rate.

Once a satisfactory BERT discriminator was trained, we leveraged it to generate messages for a provided diff. We first composed a new vocabulary out of all the words students used in their commit messages, and took any word that was used over 500 times throughout the semester. This is desirable since it both provides the model with the ability to use words specific to the domains of our assignments (such as BearMaps, proj3, autograder), and also if there are commonly used but inappropriate words used by students (such as hi, yolo, and expletives), the model has the potential to teach students to avoid using them by strongly suggesting more constructive alternatives. When then tokenized and fed into BERT each of the vocab words paired with the desired diff. The top 5 pairs which were most likely to be true pairs were selected in a beam search fashion. Following this, each of these 5 words had each of the vocabulary words appended to them individually, and were paired with the provided diff message once more, before being fed into BERT again, allowing the process to repeat.

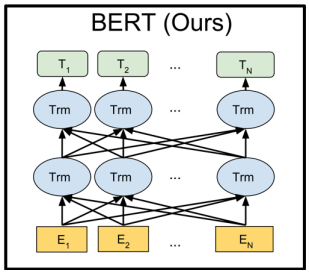

Fig. 5. The BERT architecture as shown in the original paper [1]. It is a bi-directional, multi-layer transformer.

E. GPT-2

We also attempted to use the GPT-2 from OpenAI, as GPT-2 is better suited for text generation tasks while still relying on transformers and attention; however, we found these models much more difficult to train and interact with. Although these models were not employed in this design, they seem promising and we recommend incorporating them in future work.

IV. Results

A. LSTM-Based Language Model

1) Language Model Training & Results: The language model is trained using a learned vocabulary embedding that contains embeddings for 20,000 words, including <START>, <UNK>, and <PAD> tokens. To train the model, we use a learning rate of 0.001 and a batch size of 512. We train the model for 20 epochs (each epoch equivalent to a full pass through the training set) which takes a few hours to complete on the Tesla K80 GPU provided by Google Colab. Training for longer than 20 epochs resulted in an increasing validation loss. In order to generate clear commit messages, we provide the trained model with a starting word (typically the past participle of a verb, like Updated or Added). The model then appends words one at a time to the message, and each word is determined by all previous words in the message. Generating a message using the language model requires at least one starter word, allowing us to avoid the problem of using classification to pick that word as explored in [3]. Realistically, if this tool is used by 61B students, they would be able to input one or more words at the beginning and allow the model to autocomplete a viable commit message.

- "added a convenience method to the new api infrastructure"
- "finished the test case for # version"
- "update readme with release 2"
- "changed the default value for the new api"

Fig. 6. Example messages generated by the language model.

2) Handling <UNK> Tokens: The model learns to fill in certain portions of a message with the <UNK> token for unknown words. We propose two methods for handling these tokens. First, the message with <UNK> tokens can be provided to students with the tokens replaced with blanks, allowing students to fill in these portions with terms that are likely specific to the current assignment. These terms are also unlikely to be found in the professional repositories if they are specific to 61B projects (such as BearMaps, proj3, autograder). A second way to remove <UNK> tokens is to force the generating process to only pick words from the vocabulary set that are not <UNK> tokens. We implement a version of this to generate the messages seen in Fig. 6. If the most likely word at some time is <UNK>, we choose the next most likely word instead and append it to the sentence.

B. Transformer-Based Summarization Model

The transformer architecture produced mixed results. After a few hours of training we had a training loss of 4.578874, which was a rather underwhelming result (Fig. 7). Running the transformer on the test set and sequentially taking the max of each logit gave us strings of text that barely resembled commit messages. However, most of the commit messages generated were close to unusable. We propose multiple reasons that this did not work. First, all of the repositories were not in the same human language. Even though we took the top 100 repositories on Github, there were a couple repositories which were in Chinese, which bloated our vocabulary size and forced it to teach unnatural language structures. Second, the quality of the commit messages themselves were also low quality. This is mostly because we limited ourselves to short commits and short diffs, but many commit messages, such as :memo: Writing docs, or bug fixed were uninformative and did not directly reference the code, making it very difficult

Fig. 7. Test examples for the transformer model

for the transformer to train. Finally, the way we generate commit messages for the test cases may not have been optimal. Instead of taking the highest max logit for each element in the sequence, it may have been more beneficial to keep track of the K best logits for each element, to keep options open for the transformer.

C. BERT

We performed hyper-parameter tuning on our BERT discriminator (Fig. 8). The dataset used here is a sample of 40,000 professional message/commit real and negatively sampled pairs, at a 1:5 ratio as described in the Approach section. While the F1 score generally offers a metric which balances the tradeoff between precision and recall, since the F1 score of the two smaller learning rates tested where negligibly similar, we decided that 2e-5 would be the safer option since it had better precision, and precision is unaffected by large quantities of negative samples [9], which is what we had here. Further, since the optimal learning rate matched that which was used by Google in their fine-tuning demonstration [10], we decided to adapt the remainder of their hyperparameters (batch size of 32, warm-up of 0.1). For the final BERT discriminator model, we increased the number of epochs from 0.5 to 4 and used the optimal (Google's) hyperparameter values. It achieved an accuracy of 0.8514, an F1 score of 0.5526, and a precision of 0.5547. The improvement after this additional training time was little, so longer sessions of training were not attempted.

For the BERT generator, we designed the algorithm of top-5 beam search to discriminate the most likely next words from the student vocabulary set that should match a given student commit, as described in the Approach section. Qualitatively, one example we generated a message for is shown in Fig. 9. Here, the student made a one-line change in this commit. They chose the message Tiny bug fixed, which is not a poor message choice given that this edit likely did correct a small error, but it also has potential to be improved. Running the

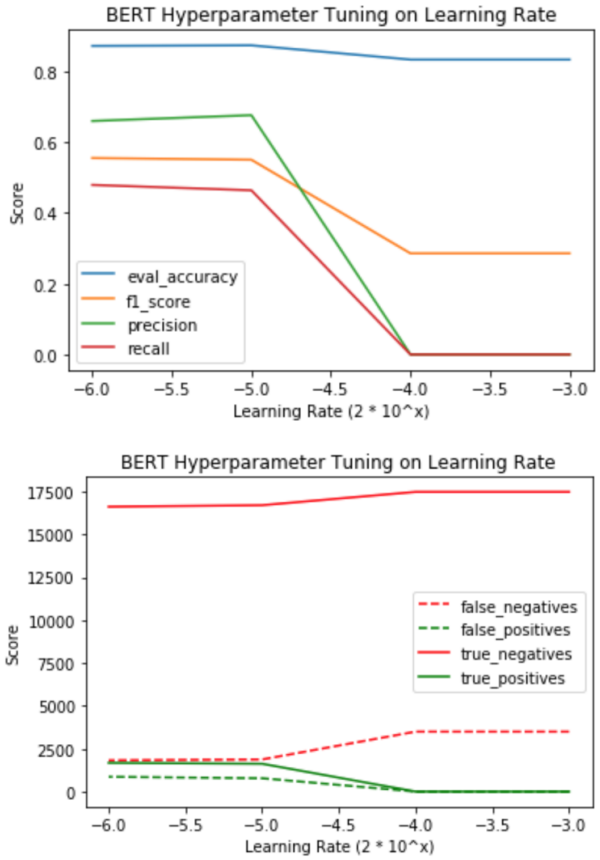

Fig. 8. Results of hyperparameter tuning on the learning rate of the fine-tuning process of our BERT discriminator.

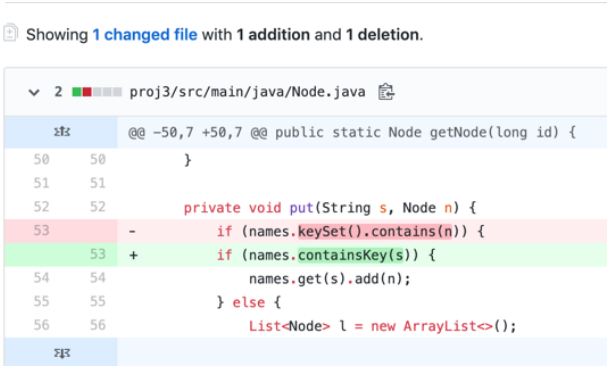

Fig. 9. Results of hyperparameter tuning on the learning rate of the fine-tuning process of our BERT discriminator.

generator, some sentences in generated over the iterations are as follows:

- ('proj3', 'node', 'proj1', 'nodes', 'proj2')
- ('proj3 ', 'nodes handle', 'nodes because', 'proj3 ready', 'nodes does')
- ('nodes does new', 'proj3 being', 'proj3 ready has', 'nodes does started', 'nodes does edits')
- ('proj3 being tiny', 'nodes does edits fixed', 'nodes does started hue', 'proj3 ready has completed', 'nodes does

new hi')

- ('proj3 ready has completed 8', 'proj3 ready has completed changes', 'proj3 being tiny tested', 'proj3 being tiny hard', 'nodes does started hue folder')
- ('proj3 ready has completed changes credit', 'proj3 ready has completed 8 ugh', 'proj3 ready has completed 8 small', 'nodes does started hue folder remove', 'proj3 ready has completed changes runtime')
- ('proj3 ready has completed changes credit string', 'proj3 ready has completed 8 small method', 'nodes does started hue folder remove ensure', 'proj3 ready has completed changes runtime statement', 'nodes does started hue folder remove merging')

The grammar of these samples is fair, and some keywords were successfully identified; this was indeed work from Project 3, and the class that is edited is called Node.java. The generator also agreed with the student that the change was tiny, and perhaps this change has something to do with iteration over a keySet of the HashTable. Unfortunately, quantitative results do not seem applicable here; we want to generate messages better than the student message, not to match it. It is also difficult to manually identify student repositories with good commit messages to improve our fine-tuning or to validate against.

V. Lessons Learned

From our results, we have learned that language models are often not stable during training, which can lead to a wide range of results. Ours, in particular, is able to avoid problems that other models often run into by requiring a starter set of words in order to provide it with the context needed to generate messages.

Additionally, the transformer architecture may require more modifications than we anticipated, in order to train on samples as small as the commit messages extracted from professional repositories. This model seems promising, though, and adjustments such as beam search, separating data by language, and training on longer messages would likely improve this approach.

Finally, our modified BERT model, while able to generate interesting suggested messages, has room to improve as well. Something that we noted was that this model appeared to be heavily influenced by the fact that the student repositories used were not necessarily adequate samples for training, given that we are attempting to improve upon student messages. A possible solution to this would be for us to use course staff repositories as they completed the student projects, giving better examples on which to train our model.

VI. Future Work

There are several possible ways to extend the current work we have done for each of the three approaches. First, we were unable to explore training an ensemble of our models for any one task due to time and computational constraints, though this could help us achieve better results on the commit message generation tasks using the RNN or transformer models. This way, each model could produce a candidate for the next word of a commit message, and the final word to be added to the message could be decided using either a vote between the models or a more complicated averaging using logits estimated by the models.

While we did not pursue constraining commit messages to different types of sentence structures as was partially explored in [3], this direction remains open for additional research. After looking through much of the dataset we collected, many commit messages tend to follow a similar sentence structure. One could develop a model to generate different types of messages for each type of sentence, and then pick the sentence with a maximal overall likelihood under the general, unconstrained language model. This would be another approach that may result in clearer, simpler commit messages.

VII. Tools

As our project involved working with text, we started building models using techniques taught in the course. This meant that we primarily used learnable vocabulary embeddings for the text while exploring transformer-based or recurrent architectures for producing commit messages. As several popular models involving deep neural networks and text perform well with learnable vocabulary embeddings, we felt that this element of our models would not require significant changes. Therefore, most of our experimentation concerned various tweaks to model architectures and hyperparameters.

However, after several updates to the initial language model and summarization model, we were unable to improve our training and converge to lower losses. Additionally, we were applying a patchwork of modifications to the outputs of these models to ensure that they were more grammatically correct and coherent; however, these techniques did not work as intended for every sample diff. We then pivoted to training proven state-of-the-art architectures on our dataset to see if we could beat the performance of our previous models. As BERT is well-documented with pre-trained weights and code freely available online, we chose to start using it to help motivate better commit messages. BERT naturally was able to learn to discriminate between relevant and non-relevant commit messages, as this is similar to one of the tasks it was original trained upon by the inventors. There seems to be some disagreement however over whether BERT can successfully lend itself to tasks involving text generation. While the authors, and Wang and Cho insist that it can by treating BERT as a Markov Random Field by masking out one or multiple tokens at a time [11], the simple experiments of others do not substantiate these claims [12], perhaps exposing the fact that BERT is bidirectional and relies on seeing into the future. Where BERT fails in generation, GPT-2 may offer an improvement, but we leave this model for future exploration.

For our development tools, we used Tensorflow to implement our models and primarily wrote Python scripts for collecting, cleaning, and filtering our datasets. Additionally, we relied on Google Colab and its associated free K80 GPU when training our models on larger batches for longer periods

of time. For short experiments or hyperparameter searches involving shallow training, we were able to train on local machines without GPUs. Pre-processed data was stored into json files for accessibility. Since tokenization of BERT features was time consuming, the discriminator's features were saved as pickle files for ease of use.

VIII. Team Contributions

- Aman Dhar (23%) - Modified language model architecture to generate commit messages, trained language model for generating commit messages, created significant portion of poster.
- Jackson Leisure (22%) - Initiated poster design, created significant portion of poster, helped modify conceptual model architecture to achieve more useful messages, helped gather student repo data, maintained contact with assigned GSI.
- Riku Miyao (24%) - Helped gather professional repo data. Preprocessed data so that it would work for the transformer. Worked on generating commit messages through the transformer architecture.
- Matthew Sit (31%) - Helped gather student repo data, performed all data pre-processing for both student and professional repos, identified student commits to use as examples to demonstrate model performance, copy-edited the poster, primary owner/developer of the BERT model aspects of the project, attempted to incorporate GPT-2 but decided not to finally pursue it due to time limitations.

References

[1] Devlin, J., Chang, M.-W., Lee, K., Toutanova, K. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv e-prints arXiv:1810.04805.

[2] Vaswani, A., and 7 colleagues 2017. Attention Is All You Need. arXiv e-prints arXiv:1706.03762.

[3] Jiang, Siyuan, and Collin Mcmillan. Towards Automatic Generation of Short Summaries of Commits. 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), 2017, doi:10.1109/icpc.2017.12.

[4] Jiang, Siyuan, et al. Automatically Generating Commit Messages from Diffs Using Neural Machine Translation. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, doi:10.1109/ase.2017.8115626.

[5] Liu, Zhongxin, et al. Neural-Machine-Translation-Based Commit Message Generation: How Far Are We? Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018, Sept. 2018, doi:10.1145/3238147.3238190.

[6] 'About Stars.' About Stars - GitHub Help, help.github.com/en/articles/about-stars.

[7] Google-Research. Google-Research/Bert. GitHub, 28 Mar. 2019, github.com/google-research/bert.

[8] Wu, Y., and 30 colleagues 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arXiv e-prints arXiv:1609.08144.

[9] 10155641495255719. What Metrics Should Be Used for Evaluating a Model on an Imbalanced Data Set? Towards Data Science, Towards Data Science, 5 Sept. 2017, towardsdatascience.com/what-metrics-should-we-use-on-imbalanced-data-set-precision-recall-roc-e2e79252aeba.

[10] "Google Colaboratory." Google, Google, colab.research.google.com/github/tensorflow/tpu/blob/master/tools/colab/bert_finetuning_with_cloud_tpus.ipynb.

[11] Wang, A., Cho, K. 2019. BERT has a Mouth, and It Must Speak: BERT as a Markov Random Field Language Model. arXiv e-prints arXiv:1902.04094.

[12] Stephen Mayhew. Can You Use BERT to Generate Text? Stephen Mayhew, mayhewsw.github.io/2019/01/16/can-bert-generate-text/.