Intelligent Code Review Assignment for Large Scale Open Source Software Stacks

***Abstract*—In the process of developing software, code review is crucial. By identifying problems before they arise in production, it enhances the quality of the code. Finding the best reviewer for a code change, however, is extremely challenging especially in large scale ,especially open source software stacks with cross functioning designs and collaborations among multiple developers and teams. Additionally, a review by someone who lacks knowledge and understanding of the code can result in high resource consumption and technical errors. The reviewers who have the specialty in both functioning (domain knowledge) and non-functioning areas of a commit are considered as the most qualified reviewer to look over any changes to the code. Quality attributes serve as the connection among the user requirements, delivered function description, software architecture and implementation through put the entire software stack cycle. In this study, we target on auto reviewer assignment in large scale software stacks and aim to build a self-learning, and self-correct platform for intelligently matching between a commit based on its quality attributes and the skills sets of reviewers. To achieve this, quality attributes are classified and abstracted from the commit messages and based on which, the commits are assigned to the reviewers with the capability in reviewing the target commits. We first designed machine learning schemes for abstracting quality attributes based on historical data from the OpenStack repository. Two models are built and trained for automating the classification of the commits based on their quality attributes using the manual labeling of commits and multi-class classifiers. We then positioned the reviewers based on their historical data and the quality attributes characteristics. Finally we selected the recommended reviewer based on the distance between a commit and candidate reviewers. In this paper, we demonstrate how the models can choose the best quality attributes and assign the code review to the most qualified reviewers. With a comparatively small training dataset, the models are able to achieve F-1 scores of 77 and 85.31 percent, respectively.**

***Index Terms*—Large-scale, Open-source, Code Review, Commit Classification, Machine Learning, MPNet**

I. Introduction

Quality attributes are the non-functional requirements evaluated by the developers and stakeholders. Software architects extract quality attributes from requirements specification to enhance the architecture design, raise stakeholders satisfaction and apply them into the software design and implementations. Identifying code check-ins related to quality attribute assist in architecture design and improving the software quality. Maintenance is an anticipated phase in the project life cycle. This phase might include project expansion or bug fixes related to certain quality attribute(s) such as security enhancement, performance improvement, test coverage or improve testability and documentation related.

Review and confirmation during the maintenance phase are also required to fulfill any required changes. Improper assignments of the reviews results in inefficiency of the software development and increase the risks of faults and vulnerabilities in them, especially large scale software systems. In this paper, we will use machine learning techniques to automate the assignment process for developers and architects based on the history of the reviewers and the quality attribute(s) extracted from the developer's commit messages. The goal of this study is to streamline the review allocation process using the traits of quality attributes. When choosing reviewers based on the application fields, the reviewers with software expertise are constrained because the functional features of software can vary significantly depending on the application fields. The quality attributes are the essential core that values the quality and consistency of software over time. Allocating reviewers based on quality attributes results in more efficient task allocation among reviewers and improves software quality in the long run.

As code check-ins include modifications or new functionalities, the system architecture may be impacted. Each code check-in can be associated with one or more quality attributes. As a result, architects will be able to define a more accurate architectural structure in which code check-in modifications are taken into account and linked to expected quality attributes.

We will build our own model using supervised learning and validate its results' using a supervised training model from a platform called Monkey-Learn. Monkey-Learn is an artificial intelligent platform offers a text analysis model using "supervised learning". Monkey-Learn model was trained with 1,000 data points to verify and compare it with our own model results. Both models will be thoroughly described later.

Researchers have used several approaches for code reviewer recommendation. We have studied a variety of research papers [1]. Numerous methods for automating source code reviewers' recommendations have been developed over time [2]–[6]. Based on the primary features they take into account and the methods they adopt to suggest code reviewers, we summarized the most pertinent existing recommendation algorithms into four groups:

- Heuristic
- Machine Learning
- Social Networks
- Hybrid approaches

A. Heuristic Algorithms

To identify the most pertinent code reviewers, traditional recommendation approaches analyze data from previous project reviews and employ heuristic-based algorithms. The

three main algorithms are Code Reviewer Recommendation based on Cross-project and Technology experience (CORRECT) [3], ReviewBot [5], and RevFinder [2] [7]. Assumptions for factors that associating the existing reviewed commits and to-be reviewed commits in the category of Heuristic Algorithms include: Expertise Assumption, Familiarity Assumption, and File location.

Expertise Assumption: In the proposed CORRECT system [3], the underlying principle is that the reviewers of the prior pull request are also suitable candidates for the current one if the previous pull request used a similar external library or technology.

Familiarity Assumption:A method called ReviewBot has been put forth by Balachandran [5]. It is a code reviewer recommendation method based primarily on the idea that lines of code modified in the pull request ought to be reviewed by the same code reviewers who previously discussed or modified those lines of code.

File Location: RevFinder [2] is based on the location of the files that are part of pull requests. The Code Reviewers Ranking Algorithm is the first component of the RevFinder approach. It employs four string comparison techniques to compare the file paths in a new pull request with all the file paths that have already been reviewed (Longest Common Prefix, Longest Common Suffix, Longest Common Substring and Longest Common Subsequence). Candidates for code reviewers are given points in this step. The number of points given to the code reviewers who previously reviewed the files increases in proportion to how similar the file paths are. The Borda count combination method is used to combine the outcomes of each of the four string comparison techniques.

B. Machine Learning Based Algorithms

Machine Learning based algorithms recommend code reviewers using a variety of machine learning approaches. They differ from the previous group primarily because they must first construct a model based on training data. The authors of "Predicting Reviewers and Acceptance of Patches" [6] employ the Bayesian Network technique to forecast reviewers and patch acceptance based on a number of features, including patch meta-data, patch content, and bug report details.

C. Social Networks Assisted Algorithms

Social networks have also been used to identify communication patterns among developers, resulting in the recommendation of candidates for source code reviews who are more likely to be similar.

By examining the social connections between contributors and developers, [4] proposed the Comment Network (CN) approach as a method for recommending code reviewers. The foundation of CN-based recommendations is the notion that it is possible to infer developers' interests from their interactions with comments. As mentioned in [4], appropriate code reviewers are developers who have similar interests to the person who created a pull request.

D. Hybrid Approaches

This group of algorithms combines various techniques (machine learning, social network analysis) to suggest code reviewers. The work in [7] is an illustration of this approach.

In the past, sentence embeddings have been the subject of in-depth research. Each approach has a unique approach to solving the problem. For instance, Skip-Thought [8] uses an encoder-decoder architecture to determine how the sentences around it relate to one another. While InferSent [**?**] makes use of a max-pooling layer and a Siamese BiLSTM network. Even though the SkipThought method is effective, InferSent consistently outperforms it [**?**]. The Universal Sentence Encoder [**?**] combines unsupervised training with SNLI training to train a transformer network. On the STS benchmark dataset, a combination of Siamese DAN and Siamese Transformer networks produced good results [**?**].

Adapting the deployment scenario is a significant challenge when deploying the current code review assignment algorithms. The assumptions in the use scenario must be met for the heuristic algorithms to be accurate. A wide range of algorithms are covered by machine learning techniques. To achieve the anticipated accuracy, the training dataset must also meet certain requirements. However, it may be deceptive and constrained by privacy concerns. Social networks offer strong pivots for reviewers selection with the rich information in their profiles, networks, and activities. For many conditions, hybrid approaches that combine the benefits of the models described above are used more frequently. Due to their complex architecture and the diversity of the developer and reviewer community, large scale open source systems pose the greatest challenge among use cases and existing techniques. The challenges couldn't be fully addressed by any of the earlier algorithms. Another obstacle is the need for non-functional quality attributes like security, modifiability, and ongoing development, where reviewers' experience and knowledge are crucial to the software stack's quality. Certain software stacks function as vital infrastructures. For example, in 5th Generation Cellular Communications software defined radio platforms like Open Air Interface (OAI) [] [**?**], srsRAN [**?**].

Thus in this paper, we presented an innovative code review assignment system that consider both functioning and non-functioning features of a commit for reviewer allocation. The proposed system addressed the challenges in large scale open source software stack. The main contribution of this study includes:

1) A proposed quality-attribute based intelligent code-review assignment system that addressed the challenges in large-scale open-source software stacks.
2) The study integrated both supervised learning-based and unsupervised learning-based algorithms with self-learning capability to improve the learning accuracy over the time.
3) Customized NLP models and transform learning scheme are developed for quality attributes abstraction from the

commit message. Both commercial platform and in-house developed models are explored and compared.
4) A modified version of the masked and permuted language modeling (MPNet) is designed and implemented aiming to determine the quality attributes for a specific commit message.
5) UMAP-based algorithm is used for identifying relative-position of the classification results of commit messages. The shortest Euclidean distance between the calculated point and the centroid point for each user is then used to characterize the reviewers specialty and assign the target commits to the matching reviewers.
6) The proposed system provides enhancement for the coherence in development process, and significantly reduces the development and maintainers cost through out the product life cycle. It allows an intelligent and efficient tasks allocation, which leads to improvement in code quality and reduce potential risks due to improper reviewing process.

The rest of the paper is organized as follows. We first introduced the backgrounds and basic components that serves as the foundation of the proposed system in Section II. We then described our proposed system in Section III, followed by the system performance assessment at Section IV. Finally, the conclusion and future work is presented in Section V.

II. Backgrounds and Basic Components

In this section, we go over the fundamentals of intelligent code review and automatic reviewer assignments. We first describe natural language processing (NLP) keyword abstraction scheme exploring existing commercial NLP platform that can be leveraged. We then discussed models that uses Transformers in NLP. To build in-house customizable classification model, we have developed transfer learning model based on a SBERT model named MPNet [**?**]. At last, we presented the Clustering methods used for reviewer matching to build our proposed systems.

A. NLP Based Quality Attributes Annotation and Classification Using Monkey-Learn platform

Monkey-Learn is a user-friendly artificial intelligence platform that offers a ready-to-train text analysis model without any coding skills. This platform is based on supervised learning which requires user annotation. Due to its many cost-effective pricing options for product teams and developers and high quality performance, Monkey-Learn is widely used in the industry [9].

In this study, We have utilized the text analyzer's free trial plan, which offers 1,000 training data and 1,000 outputs each month and compared with our in-house classification models.

B. Transformers Deep Learning Model in NLP

Transformer models are useful in transfer learning from a large-scale LM (Language Model), which have been pre-trained on a significant amount of text for conceptual tasks, such as sentiment analysis and even predictive analysis. A transformer is a deep learning model that uses the self-attention process and weights the importance of each component of the input data differently [**?**]. The goal of transformer architecture is to account for the distant relationships between words with ease while solving sequential problems.

In contrast, alternative models which are available for creating information-rich representations of sentences and paragraphs are sentence transformers. These models are trained using a number of techniques, but the unsupervised pre-training of a transformer model using techniques like masked-language modeling is the first step.

Because BERT [**?**] fails to account for dependency among predicted tokens we use MPNet for transfer learning. In order to make the most of the dependencies between predicted tokens, MPNet employs permuted language modeling. It also uses auxiliary position information as input to help the model recognize complete sentences, which lowers position discrepancy. The primary advantage of MPNet over BERT is that it uses more information when predicting a masked token, resulting in better representations while learning and less discrepancy with the downstream tasks. Since BERT ignores the relationships between predicted tokens, XLNet [10] introduces permuted language modeling (PLM) for pre-training to address this issue. However, XLNet suffers from position discrepancy between pre-training and fine-tuning because it does not fully exploit the position information contained in a sentence. MPNet inherits the benefits of BERT and XLNet while avoiding their drawbacks. In contrast to MLM in BERT, MPNet uses permuted language modeling to take advantage of the dependencies between predicted tokens. It also uses auxiliary position information as input to help the model perceive a full sentence, thereby minimizing position discrepancies. According to experimental findings, MPNet significantly outperforms MLM and PLM on these tasks and outperforms earlier state-of-the-art pre-trained methods (such as BERT, XLNet, and RoBERTa) when used in the same model setting [**?**].

III. System Description

The objective of the proposed system is establish an self-learning, and self-correct platform for intelligently code review assignments to match between a commit based on its quality attributes and the skills sets of reviewers. The primary targeting software stacks are large scale open source software stacks where the modifiability, reliability and security significantly influence the quality if software stack or delivered system. The challenges of code review assignment in such systems include cross functioning designs and collaborations among multiple developers or teams, which requires the reviewers' expertise in both domain knowledge and non-functioning software architectures.

Quality attributes servers as the connection among the user requirements, delivered function description, software architecture and implementation through put the entire software stack cycle. Thus, in our proposed system, quality attributes are classified and abstracted from the commit messages and

based on which, the commits are assigned to the reviewers with the capability in reviewing the target commits.

In Fig.1, we have described the overall system design and implementation, including multiple independent modules and interfaces bridging the modules. We start by selecting OpenStack as our preferred software stack. Next, we divide the collected data samples into commits and reviewer histories. Preprocessing is done on the data obtained from the commit samples for training and testing. The MonkeyLearn and the MPNet models receive the annotated data samples as input. For every commit, these models predict the quality attribute. For dimensionality reduction, the output is fed to the UMAP model along with the commit reviewer data. The plot containing the centroid coordinates for the top reviewers and each commit sample is created using the 2-D points of the output. After that, we determine the Euclidean distance to compare the reviewers to the commits. In order to assign the reviewers in an intelligent manner, we take into consideration external factors like domain expertise, team members, etc.

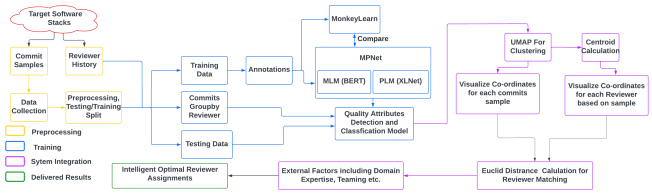

Fig. 1. System Overview

A. Data Collection

The code review process of the OpenStack is based on Gerrit[1], collaborative code review framework allowing developers to directly tag submitted code changes and request its assignment to a reviewer. Generally, a code change author opens a code review request containing a title, a detailed description of the code change being submitted, written in natural language, along with the current code changes annotated. Once the review request is submitted, it appears in the requests backlog, open for reviewers to choose. Once reviewers are assigned to the review request, they inspect the proposed changes and comment on the review request's thread, to start a discussion with the author.

We mined code review data using the `RESTful API`[2] provided by Gerrit, which returns the results in a `JSON` format. We used a script to automatically mine the review data and store them in `SQLite` database. All collected reviews are closed (i.e., having a status of either '*Merged*' or '*Abandoned*').

B. Experiment Setup and Annotation

First, we established the tags of annotation on both Monkey-Learn platform and our models. The following tags have been used: Documentation related, Performance improvement, Test coverage or Improve test-ability, Security enhancement, Improve Availability, Improve usability, Achieving interoperability, Improving modifiability.We began the training process after identifying the tags. With 1,000 check-in data, we trained the model with manually assigned quality attribute(s). Monkey-Learn tracked our behavior to identify a pattern that can be used to categorize the data going forward. The model carried out this process by identifying words that are related to the same tagged quality attribute in the classified check-ins.

The model is now ready to receive an input, process it, and produce an output. Based on the identified patterns during the training process, Monkey-Learn will produce a list of words associated with each tag. The input text will be automatically analyzed by the model and assigned one or more tags.

The model has phases for building, training, and running. Duplication is removed from the data during the preparation phase. No ambiguous commits were removed, and no typos or grammatical errors were fixed. Eight tags were defined, and the text may fit into one or more of these groups: Documentation related, Performance improvement, Test coverage or Improve test-ability, Security enhancement, Improve Availability, Improve usability, Achieving interoperability, and Improving modifiability. We uploaded 1,000 commits for training purpose. The trained data includes the subject and check-in code descriptions, which are required to extract the quality attribute. To create a classification model, the monkey-learn model is fed training data that consists of pairs of code-check-ins (title and description) and tags/labels (security, availability, usability, etc.). During the training phase, each of the 1,000 commits is assigned one or more labels. The machine learning model can begin to make accurate predictions once it has been trained with enough training samples and will determine which tags are associated with each input. But it is imperative to use data other than the trained data to accurately test whether the model is working properly.

In Fig.2, we find that most of the data were related to documentation, improve availability and Testing.

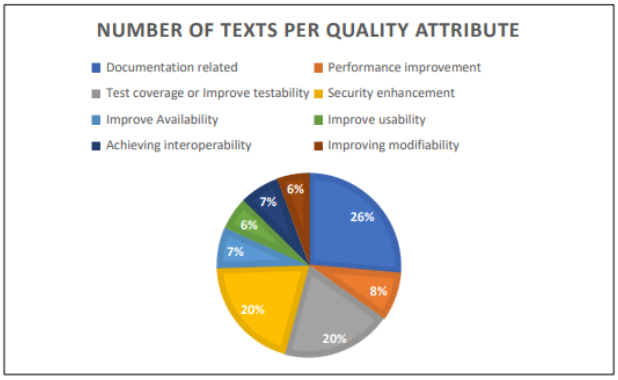

Fig. 2. Texts percentage per each tag for 1K data

In Fig.3, a few tags were having less training data which made it more challenging to have correct output related to

[1]https://www.gerritcodereview.com/

[2]https://gerrit-review.googlesource.com/Documentation/rest-apichanges.html

them while testing.

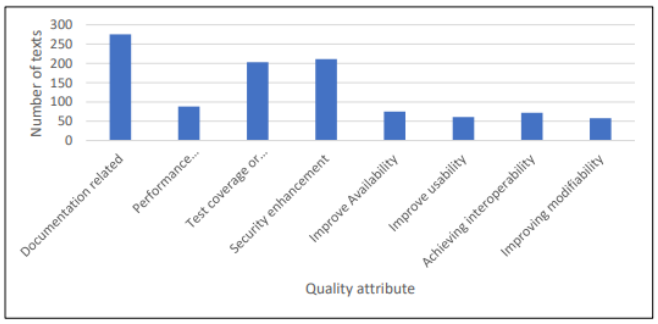

Fig. 3. Number of texts per each tag

Several commits included a description indicating that development bugs were fixed. Those commits were skipped and did not contribute to the 1,000 tagged data. We were unable to assign them to a specific quality attribute.

Fig.4, shows overall statistics for Monkey Learn model with an Accuracy of 71% and F1 Score of 77%.

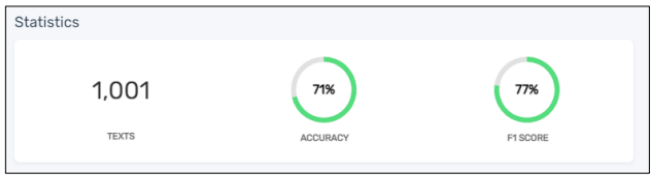

Fig. 4. Monkey Learn model statistics

C. Development of SBERT Model for Quality-Attribute Based Classification and Clustering

SBERT employs a Siamese architecture wherein it comprises two similar BERT architectures and shares their weights. It processes two sentences [**?**] as pairs during training. Let us assume that in SBERT, sentence A is fed to BERT A and sentence B is fed to BERT B. Sentence embeddings from each BERT model are combined. While the original study paper evaluated several pooling strategies, they discovered mean-pooling to be the most effective option. Mean pooling is a method for generalizing features in a network which operates by averaging the sets of characteristics from the BERT network.

We now have two embeddings, one for sentence A and one for sentence B, after the pooling is complete. The two embeddings are combined by SBERT during model training, after which a SoftMax classifier and a SoftMax-loss function [**?**] are used to train the model. The two embeddings are then compared using a cosine similarity function, which generates a similarity score for the two sentences, at inference, or when the model starts predicting.

The objective function for classification [**?**] uses the sentence embeddings u and v, and concatenate them with the

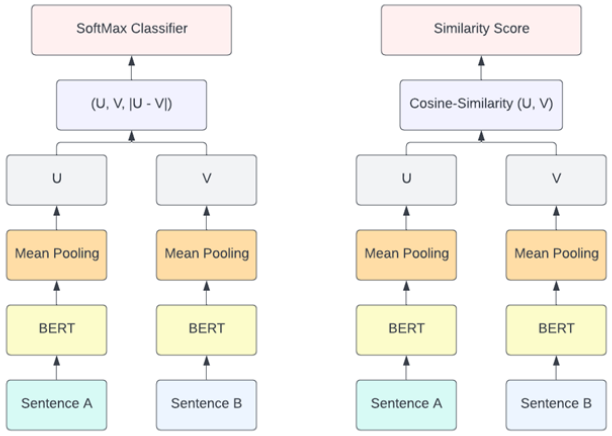

Fig. 5. SBERT Architecture

element-wise difference and multiply them by the trainable weight

$$W_t \in R^{3n \times k}$$

o = softmax

$$(W_t(u, v, |u - v|))$$

where n denotes the size of the sentence embeddings and k denotes the number of labels Cross-entropy loss is optimized. Fig.6 depicts the structure of both of the model.

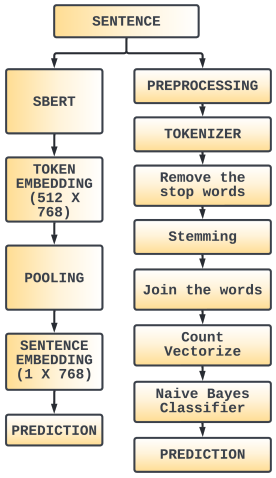

Fig. 6. The flowchart for the models

Our code reviewer recommendation model, using historical code reviews, examines how frequently the recommended reviewers were the actual reviewers for each quality attribute. However, we are going to assume that the reviewer assignment in the past was correct.

TABLE I
HYPERPARAMETERS

Name	Value
MAX LENGTH	512
TRAINING BATCH SIZE	32
VALIDATION BATCH SIZE	32
EPOCHS	40
LEARNING RATE	1e-05

IV. SYSTEM PERFORMANCE

We get an accuracy of 79.291 percent and an F-1 score of 85.31 percent when we use the MPNet-based model to make predictions. The accuracy rises to 84.89 percent when the first two tags predicted by the model are considered, and to 87.5 percent when the first three tags predicted by the model are considered.

TABLE II
METRICS

Name	Value
Accuracy	0.79291
Precision	0.8824
Recall	0.8257
F1 score	0.8531

For each of the quality attributes, the model determines the confidence levels. The values of the prediction vector and the value of the quality attribute for each row are chosen in order to further visualize the results. These values are then fed into the UMAP and T-SNE algorithms as shown in Fig.7 and Fig.8 respectively, which produce the 2-D coordinates for the commit messages. An 8-D vector is reduced to a 2-D vector in the process. We discovered that the UMAP algorithm produces better clustering of our data when we compare the visualizations. As a result, we assign reviewers based on the coordinates generated by the UMAP clustering.

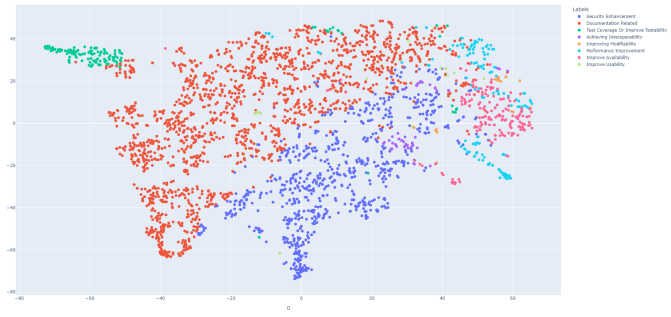

Fig. 7. T-SNE clustering

The centroid points in Fig.9 are then calculated for the top twenty reviewers who are not part of a team of multiple reviewers. This makes it easier for us to assign commits to reviewers for review. The Euclidean distance is used to predict which commit should be assigned to those reviewers. The distribution is depicted in Fig.10

Fig. 8. UMAP clustering

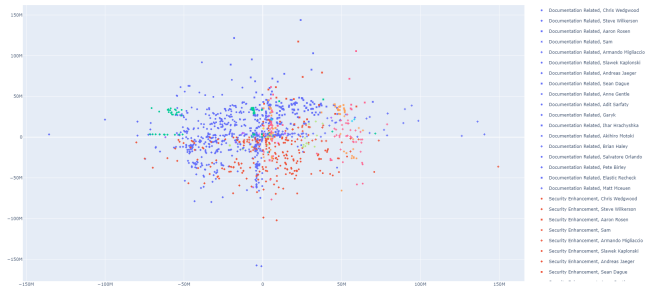

Fig. 9. Centroid for the users

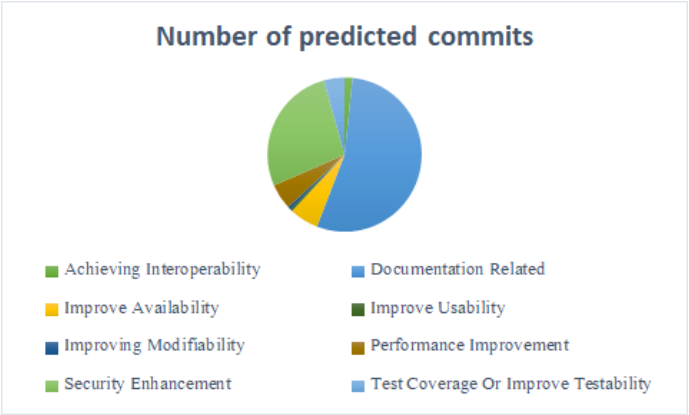

Fig. 10. Number of predicted commits

V. CONCLUSION

Without code review, the modern software development process cannot be accomplished. It ensures that the entire process is coherent, and improves the code quality, while lowering the overall cost of the software. Finding the ideal code reviewer is challenging. This can take a lot of time, and improper task distribution could end up costing the project a lot of money. Additionally, the reviewers' workloads might not be distributed equally, which could put more of a burden on a select few. In order to find the best candidate for reviewing open pull requests, we have come up with our own custom model.

To address the issues with large-scale open-source software stacks, we proposed an intelligent code-review assignment system that is based on quality attributes. For increasing the learning accuracy over time, the proposed platform combined supervised learning-based and unsupervised learning-based algorithms with self-learning capabilities. A modified version of the MPNet model was created and put into practice with the intention of identifying the qualities of a given commit message. After that, a UMAP-based algorithm was used to determine the relative position of the commit message with respect to the classification results. The reviewers' expertise are then predicted and the target commits are assigned to the matching reviewers using the shortest Euclidean distance between the calculated point and the centroid point for each reviewer.

At the end of this study, we found that the two models performed admirably at classifying code commits according to their quality attributes. With the data that has been given to it, the proprietary Monkey-Learn model functions well, but it has a number of restrictions. For instance, it was unable to provide us with the probabilities for each of the eight quality attributes. Because of this, it was difficult to determine the two-dimensional coordinates for the commits. The MPNet-based model, however, performed better in terms of accuracy and flexibility. It could provide us with the eight-dimensional vector values to calculate the coordinates for reviewer assignment, and was 11.7% more accurate than the proprietary solution. When considering the F1 score, the MPNet-based model outperformed the monkey-learn model by a margin of 10.8%.

VI. Future Work

In this paper, we show that the MPNet-based transformer model can predict the reviewers who will be able to effectively address the pull request and can account for the complex interactive patterns between the entities in the code review ecosystem. While the data from the OpenStack code repository shows promise, we think that by training the model on the data unique to any other repository will be useful in making recommendations. The model can be used by any project to streamline their workflow because it is quite generic and can be trained on any dataset made available to it. Here, we primarily focused on the relationship between the commit messages in the OpenStack code repository and the quality attributes. The files and directory where the changes were made were not taken into consideration. The model may learn intricate patterns in the data by keeping track of those features, increasing the accuracy of selecting the reviewers. Additionally, we think that a thorough examination of the hyperparameters would result in greater accuracy. In our future work, we want to examine the effect of these parameters on the overall behavior of the model.

References

[1] Jiyang Zhang, Chandra Maddil, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhawar, Kim Herzig, and Arie van Deursen, "Using Large-scale Heterogeneous Graph Representation-Learning for Code Review Recommendations," in *arXiv:2202.02385v2*, vol. 2657. CEUR-WS, 2022, pp. 1–9.

[2] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. I. Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, 2015.

[3] M. M. Rahman, C. K. Roy, and J. A. Collins, "CoRReCT: Code reviewer recommendation in GitHub based on cross-project and technology experience," in *Proceedings - International Conference on Software Engineering*, 2016.

[4] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?" *Information and Software Technology*, vol. 74, 2016.

[5] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings - International Conference on Software Engineering*, 2013.

[6] G. Jeong, S. Kim, and T. Zimmermann, "Improving code review by predicting reviewers and acceptance of patches," *Research on Software*, 2009.

[7] J. Jiang, J. H. He, and X. Y. Chen, "CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation," *Journal of Computer Science and Technology*, vol. 30, no. 5, 2015.

[8] R. Kiros, Y. Zhu, R. Salakhutdinov, R. S. Zemel, A. Torralba, R. Urtasun, and S. Fidler, "Skip-thought vectors," in *Advances in Neural Information Processing Systems*, vol. 2015-January, 2015.

[9] Y. Ren, J. Fan, and B. Zhou, "Autolibrary - a personal digital library to find related works via text analyzer." [Online]. Available: https://dsc-capstone.github.io/projects-2020-2021/reports/project_39.pdf

[10] P. Yang, Y. Xiao, M. Xiao, and S. Li, "6G Wireless Communications: Vision and Potential Techniques," *IEEE Network*, vol. 33, no. 4, 2019.