Mining Mutations from Public Python Projects

Adil Botabekov (20180806), Yehyoung Kang (20110162), Nguyen Tien Dat (20190883),

Nabila Sindi (20180744)

KAIST

Daejeon, South Korea.

Author Note

This is the final report for our CS453 Project.

Mining Mutations from Public Python Projects

When creating software, it's important to cover different edge cases. A method to test whether the software is of good quality is by software testing. One of the most commonly used methods is mutation testing. Mutation testing is a type of software testing where some snippets of the code are changed in order to see if certain test cases generate an error. This is to ensure the quality of test cases. However, mutation testing relies heavily on the quality of mutation operators. We cannot depend on common mutation operators since they may not be representative of actual bugs. However, it's hard to come up with enough mutation operators manually. We propose to create mutation operators from automating mutation mining from real-world bugs. Our target language is Python, so we only include code changes done in Python. The real-world bugs will be bug fixing commits from developers on Github. After mining and pre-processing the codes, we will cluster and categorize the bug fixes into certain categories to be able to find prominent bugs listed by developers. This ties in with our motivation of analyzing what kinds of errors are found in deployed code. Previous have tried mining mutations from bug fixes. Our approach was inspired by the paper by Tugano et al. [2], where they introduced an approach on how to mutate code from bug-fixes. The paper by Beller et al. [1] explored the pipeline further by focusing on the Java language. We would like to utilize the approach used by both but specify it for Python written code. To summarize, the contributions of our project is:

- Introducing a pipeline of mining bug-fixing commits from real-world commits, vectorizing and clustering them, as well as showing a visualization of the result.
- Results of silhouette scores of different combinations of vector sizes and cluster sizes.

APPROACH

The following are methods and steps done in the project:

Extracting commit data from Google Big Query

We utilized Google BigQuery for mining the commits. We mined from one of Bigquery's open datasets, Github Activity Data [3], which includes activities from 3M open souce libraries. We set some requirements on the kinds of Github commits we mine, we included only commits with languages that include Python, ones that change more than 1 file, and with messages containing keywords such as bug, fix, issue, or error. We ended up with 3.34 million commits, with each commit containing its repository name, SHA-1 hash of the commit, SHA-1 hash of the

parent commit, and paths of all the files changed. The query was written using SQL and ran through Google's BigQuery platform.

Crawling and Chunking

Crawling. To crawl and preprocess large amounts of data in parallel, we split the query result into multiple chunks, with each chunk containing approximately 100,000 rows (commits). For every commit, we used GitHub's web API [4] to download the contents of each Python source file before and after the commit. We ignored files that were no longer available because the source repositories were removed.

Abstraction. Preprocessing To reduce the vocabulary (number of words) required to describe the code, by replacing identifiers and constants with special identifiers.

- Identifiers: IDENTIFIER_0, IDENTIFIER_1, …
- String literals: STR_X
- Integer literals: INT_X
- Float literals: FLOAT_X
- F-string literals: F_STR_X

Note that Python f-strings are technically not constant literals since they can contain arbitrary expressions. However, we found that f-strings pose a major challenge to tokenization and refactoring as they can contain arbitrary newline characters. Thus, we replaced all f-strings with special identifiers to facilitate tokenization.

We excluded common constants and variable names, or "idioms", that frequently occur in Python code from being replaced. These idioms were extracted from 6 large and popular open-source Python repositories on GitHub, as they were deemed to be representative of most Python projects. These include software, such as Django, Flask, Keras, as well as repositories containing instructional materials, including The System Design Primer and The Algorithms Python.

Note: The reference paper extracted idioms from the source dataset. This was not possible for us, however, because we wanted to crawl and preprocess multiple chunks in parallel. Thus, we needed a collection of idioms without obtaining the source code.

Thus, we chose 6 large and popular open-source Python projects that we believed to be representative of actual Python code, and we counted the frequency of all variables and constants.

Embedding/Vectorizing

Before we can start clustering our mined commits, we need to first find a way to represent them as numerical vectors since the clustering algorithms cannot work with raw strings. In more detail, from each pair of buggy code and fixed code, we use GumpTree [5] to generate a list of edit actions A that transform fixed code into buggy code. Edit actions are string tokens, like “delete-node”, “move-tree”, "update-node",... Therefore, A has the form like this: A = [“delete-node”, “update-node”, “update-node”, …] . We then collect a list of edit actions from every pair of buggy and fixed code, and use them as an input to train the Gensim’s Doc2Vec model [6]. After training is finished, given a list of edit actions as input, the trained model could generate a corresponding vector of the fixed size for each datapoint. For example, if A = [“delete-node”, “update-node”, “update-node”...] is given as input, the model could output a vector v like this: v = [0.12 0.345 0.5 …]. Thus, from any pair of buggy and fixed code, we can generate a corresponding vector to represent that pair, and this vector would be used later for the purpose of clustering. Our Doc2Vec model is trained such that if A and B are 2 similar lists of edit actions, then the vectors generated by our model for A and B would be close to each other. The benefit of training a model to generate embedding for data is that we can learn similarities directly from the data.

Clustering

The rationale behind using clustering is to group similar mutations together and then sample from each cluster in a balanced way (to produce a diverse set of mutation operators and not just many of the same type).

For clustering, we used two algorithms, K-Means and Gaussian Mixture Model. The reason for choosing K-Means is that it’s a simple and powerful algorithm that is also used by the authors of the reference paper. The reason for choosing Gaussian Mixture Model is because, as opposed to K-Means, it takes into account the variance of the data and not just the mean of it. It’s

also a soft clustering algorithm that allows for slightly overlapping clusters to occur, which isn't the case for K-Means that is considered to be hard clustering.

Thus, we can compare two fairly different algorithms and settle for the more suitable one in the end.

The number of clusters used were 2, 3, 4, 5, 6, 8, 16.

Visualizing

To assess the effectiveness of our clustering we need to visualize our data. One way to visualize clusters is to plot the embeddings and color the points based on what cluster they belong to. However, our data is not 2-dimensional.

To visualize our data, we have to first run it through a dimensionality reduction algorithm/model. For these purposes we have used t-SNE and PCA algorithms. It is recommended to use them together, PCA being the first. We have tried several combinations and indeed settled for using PCA first and then running the PCA output through t-SNE to produce optimal visualization.

Sampling

After clustering is done, we need to sample from each cluster in a balanced way. We have come up with our own heuristic/algorithm that is as follows:

i) We build a graph for each cluster in which each node represents a single datapoint (i.e., a mutation).

ii) Then, we calculate the cosine distance between all pairs of nodes

iii) For pairs of nodes with cosine distance larger than a certain threshold, an edge is added

iv) The node with the largest degree (no. of edges connected to it) is then chosen as a representative node from the cluster and is sampled

That process is repeated for each cluster and the number of samples then equals the number of clusters.

The rationale behind this algorithm is largely intuitive. A point that is similar to as many other points in the cluster as possible should be the most representative point of that cluster.

Silhouette Score Evaluation

After clustering is done, we need to sample from each cluster in a balanced way. We have come up with our own heuristic/algorithm that is as follows: Silhouette Coefficient or silhouette score is a metric used to calculate the goodness of a clustering technique. Its value ranges from -1 to 1. 1 means that clusters are well apart from each other and clearly distinguished, 0 means that clusters are indifferent, or we can say that the distance between clusters is not significant, while -1 means clusters are assigned in the wrong way. This is a visualization of Silhouette score:

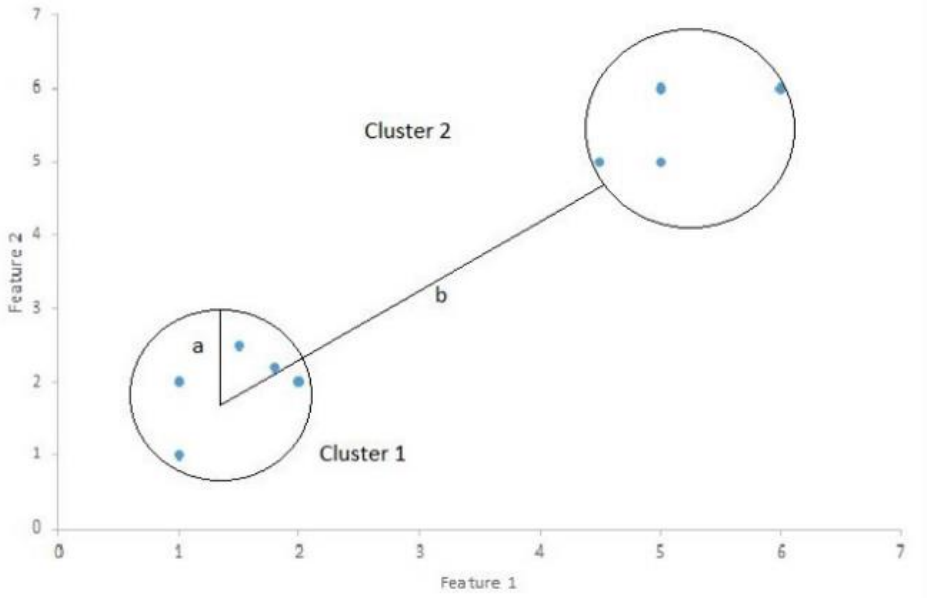


Fig 1. Visualization of Silhouette Score.

The formula for calculating: Silhouette Score = (b-a) / max(a,b), where a = average intra-cluster distance and b = average inter-cluster distance.

RESULTS

Silhouette Score

We fine-tune two parameters, which are numbers of clusters and the vector size, and record the Silhouette score for each setting. These are the results:

Fig 2. Results of Silhouette scores of different parameters.

For vector size = 5:

Number of clusters	Silhouette score
2	0.36445
3	**0.42311**
4	0.38316
5	0.34782
6	0.35342

For vector size = 20:

Number of clusters	Silhouette score
2	0.35168
3	0.41482
4	0.39859
5	0.37131
6	0.34029

For vector size = 40:

Number of clusters	Silhouette score
2	0.35446
3	0.41928
4	0.40131
5	0.37247
6	0.35874

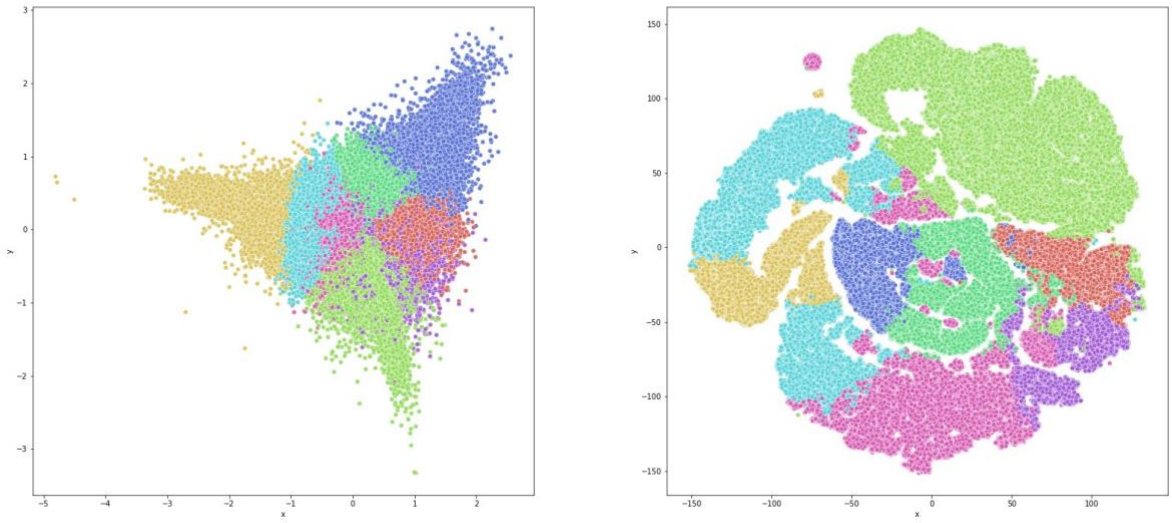
As mentioned earlier, a score between -1 and 0 is a bad score, and a score closer to 1 means a good quality of clustering. For all settings in our experiment, the scores range from 0.34 to 0.42, therefore we think that the quality of our clustering technique is not bad. Among all settings, we found the best Silhouette score at vector size = 5 and number of clusters = 3.

Visuals

Below are two visualisations of K-Means clustering with 8 clusters. On the left, the PCA algorithm was used for dimensionality reduction and on the right a combination of PCA and t-

SNE was used. As predicted earlier, PCA alone does not produce adequate dimensionality reduction to manually assess how well clusters are formed.

Fig 3. (Left) PCA and (Right) t-SNE and PCA visualizations of K-means clustering with 8 clusters.



Hence, a combination of both algorithms was used afterwards. For Gaussian Mixture Model, here is clustering visualized for 5, 8 and 16 clusters from left to right respectively.

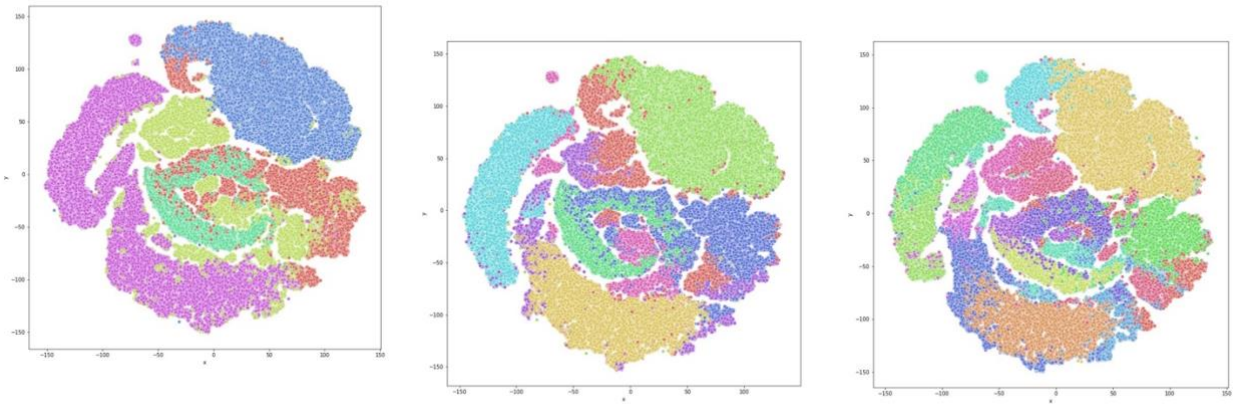


Fig 4. Gaussian Mixture Model for 5, 8, 16 Clusters respectively.

It can be seen that the clustering generally works better here for a smaller number of clusters. There are less awkward overlaps and the clearly defined blobs are more uniformly

colored for K = 5 as opposed to 8 or 16. The same can be said for K-Means clustering (K = 5, 8 and 16):

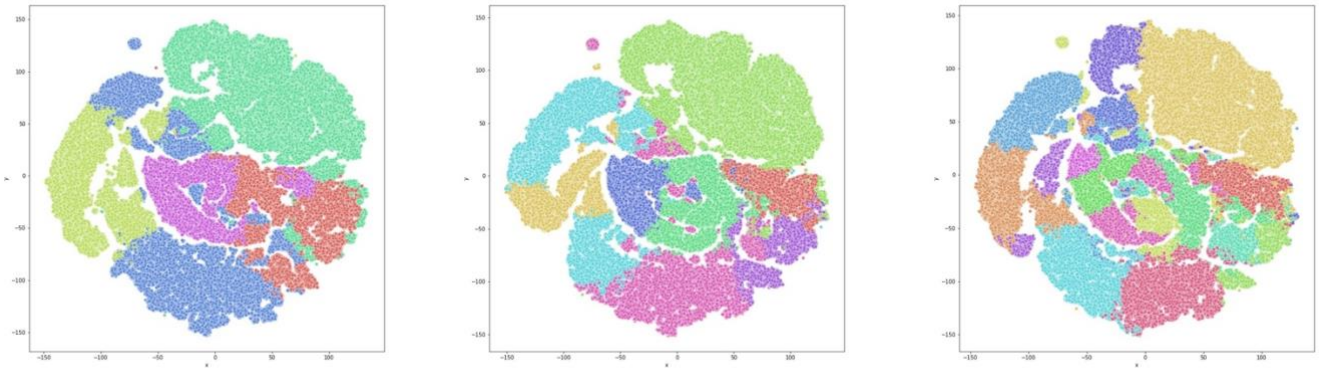
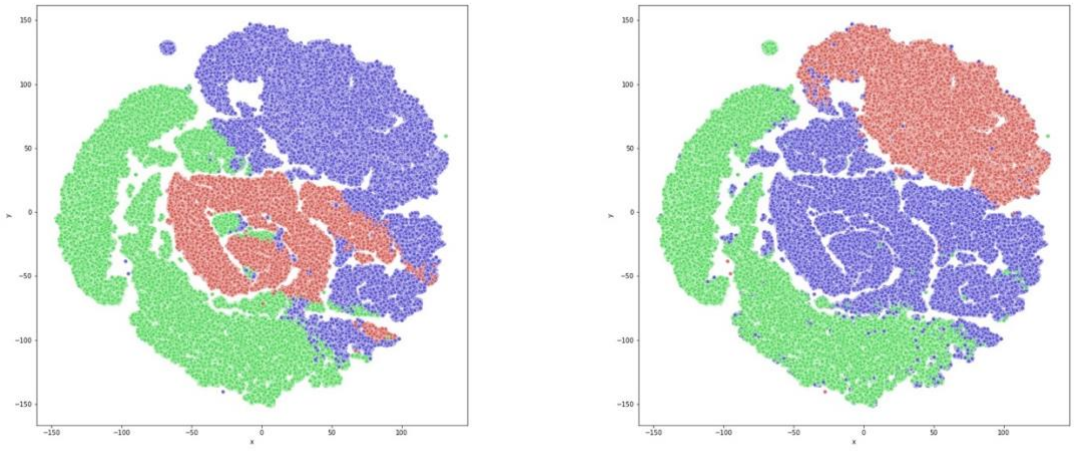


Fig 5. K-means clustering for K = 5, 8, 16.

However, as it was also predicted, since K-Means clustering doesn't take into account the variance of the data, it forces the clusters to be more circular shaped. Since it's also a more "hard" clustering algorithm, the blobs that should be uniformly colored are instead abruptly split into several colors. Thus, we believe that the Gaussian Mixture Model is a more suitable

Fig 6. (Left) K-means, (Right) Gaussian Mixture.



Algorithm for our purposes and thus we will conduct sampling using Gaussian Mixture. It is also fair to note that Silhouette evaluation shows the best score for K = 3. It can be

confirmed by visualization as well since for K = 3 both clustering algorithms produce the most visually reasonable color coding.

Samples

After running the sampling script for 8 GMM clusters, we were presented with the following code snippets.

BEFORE

```
def IDENTIFIER_0 ( self ) :
    self. IDENTIFIER_1( )
    IDENTIFIER_3 . IDENTIFIER_2 ( STR_0 )
    print ( IDENTIFIER_4 ( IDENTIFIER_6 . IDENTIFIER_5 () ) )
```

AFTER

```
def IDENTIFIER_0 ( self ) :
    self . IDENTIFIER_7 ( )
    IDENTIFIER_3 . IDENTIFIER_2 ( STR_0 )
    print ( IDENTIFIER_4 ( IDENTIFIER_6 . IDENTIFIER_5 () ) )
```

Fig 7. Snippet 1 (edit actions: [“update-node”])

As we can see, this snippet only features a change in identifier name. This most likely won’t result in a well-generalizable mutation, although it is atomic and somewhat reasonable if we can make such a mutation compilable (so it doesn’t throw a NameError).

BEFORE

```
def IDENTIFIER_0 ( IDENTIFIER_1 , IDENTIFIER_2 ) :
    f = c . IDENTIFIER_3 ( STR_0 % ( IDENTIFIER_1 , IDENTIFIER_2 ) )
    return f is not 0
```

AFTER

```
def IDENTIFIER_0 ( IDENTIFIER_1 , IDENTIFIER_2 ) :
    f = c . IDENTIFIER_3 ( STR_0 % ( IDENTIFIER_1 , IDENTIFIER_2 ) )
    return f != 0
```

Fig 8. Snippet 2 (edit actions: ["insert-node", "delete-node"])

```
def IDENTIFIER_0 ( self , name , value ) :                    BEFORE
    if value == None :
        value = ''
    value = str ( value )
    self . IDENTIFIER_1 . append ( ( name , value ) )
    return


def IDENTIFIER_0 ( self , name , value ) :
    if value is None :
        value = ''                                            AFTER
    value = str ( value )
    self . IDENTIFIER_1 . append ( ( name , value ) )
    return
```

Fig 9. Snippet 3 (edit actions: ["delete-node"])

As we can see, the mutations from snippet 2 and 3 are actually valid. They are not only atomic, but they look like actual compilable and survivable mutants. However, these mutation operators are not too different from regular operators that existing mutation testing tools can produce. Despite this, we think that the results can be considered successful due to the validity of the produced samples.

DISCUSSION

Current Limitations

Despite some success in the mining, clustering and sampling of mutation operators from public GitHub repositories, there are still significant challenges and roadblocks that we could not yet overcome due to limited time, resources and knowledge. They are as follows:

1. We could not find an automated approach to evaluate the quality of our mined and sample mutations. One way to do that would be to try and create a mutation testing tool and see how many real life test cases these operators can survive, but it would take a lot more time to achieve that.
2. Hard to evaluate the usefulness of the clustering and sampling stage. It is likely that clustering wasn't meaningful, especially because out of ~500,000 data points there surely must be a lot more "types" of bugs than 3 or 5, which are the ideal number of clusters as found by our technique. It seems that using a much more simple deterministic technique

such as more filtering (e.g., eradicating mutations that only change identifier names, etc) would be a better alternative path to follow as opposed to clustering.

3. Hyperparameters for dimensionality reduction, clustering, sampling and even the embedding model were most likely not optimal. We have noticed, especially with dimensionality reduction, that hyperparameters can greatly affect the results of our experiments (e.g., for t-SNE it was the perplexity parameter). However, because some stages took a lot of computational resources and time (for sampling on the whole dataset, more than 20 hours), it was not viable to try different hyperparameters which might have led to suboptimal results.

Possible Improvements

Despite the limitations, we are confident that this project could be the basis of future projects, here are some possible directions of improvement:

1. Expand AST processing for Python 2 code

While preprocessing, we discovered that a non-trivial amount of projects use Python 2 syntax, such as `print` statements. However, Python 3's `ast` module can only process Python 3 syntax. Since we used the `ast` module to manipulate Python code, we could not process such code, and instead chose to ignore them. Using an alternative AST implementation would enable parsing and preprocessing Python 2 code.

2. Utilize normalized before and after code.

In this project, we only built embeddings based on the list of edit actions for each data point. However, the normalized before and after code wasn't actually used in any way (apart from acquiring the list of edit actions). One way to use them would be to find a way to measure some kind of a distance between them in a comprehensive way. For example, get an embedding of before code and an embedding of after code and find a cosine distance between them and use it. In such a case, normalization (replacing all identifiers with generic ones) would actually help and be useful.

3. Tune hyperparameters
4. Train an ML model that given fixed code as input, the model can learn to generate buggy code as output.*
5. Use beam search to look for the best generated examples and weed out uncompilable mutants

*As a result of our work, now we have produced about 470,000 pairs of buggy-fixed code written in the python language which can be enough to train a reasonably robust deep learning model.

For more detail, we intended to use a model that is based on an RNN encoder-decoder architecture, which is commonly adopted in Machine Translation. This model comprises two main components: an RNN Encoder that encodes a sequence of terms x into a vector representation, and an RNN Decoder that decodes the representation into another sequence of terms y.

After the training finishes, given any piece of working python code as input, we can use our trained model to generate the buggy version of that code. So, our trained model could be developed further to build a complete mutation testing tool for python code. One advantage of this method is that the model can automatically learn mutants from faults in real programs, therefore during inference, the model could generate more realistic bugs than the bugs designed manually by humans.

However, perhaps the best alternative improvement would be to use the mined and preprocessed data and filter it with more classical and deterministic ways rather than using the Data Science-y techniques outlined in this report. For example, we could manually look at some samples and infer the types of bugs/commits to filter out. One such bug we found is just identifier re-naming, which is very tricky to make into an actual mutation operator. Therefore, an alternative path like this might be also reasonable.

CONCLUSION

As we have seen, mining mutation operators from public GitHub repositories has proven itself to be fruitful to a great extent and that actual valid and maybe even survivable mutants can be mined from real world bugs. Of course, the techniques used in the latter parts of this project to select the most suitable mutations can be substituted with any other reasonable heuristic. The important conclusion here is that public bug fixes actually do contain helpful mutation operators and this path is worth exploring further.

References

[1] M. Beller et al., "What It Would Take to Use Mutation Testing in Industry—A Study at Facebook," 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2021, pp. 268-277, doi: 10.1109/ICSE-SEIP52600.2021.00036.

[2] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White and D. Poshyvanyk, "Learning How to Mutate Source Code from Bug-Fixes," 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 301-312, doi: 10.1109/ICSME.2019.00046.

[3] Github. Google BigQuery Public Data Github Activity Data. Version 9/14/19. Accessed from https://console.cloud.google.com/marketplace/product/github/github-repos?project=cs453-spring2021.

[4] Github, “Github REST API,” Version 3. [Online] Available: https://docs.github.com/en/rest.

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, Martin Monperrus. Finegrained and Accurate Source Code Differencing. Proceedings of the International Conference on Automated Software Engineering, 2014, Västeras, Sweden. pp.313-324, ff10.1145/2642937.2642982ff. ffhal-01054552f

[6] Gensim. Doc2Vec model. Version 4.0.0. [Online]. Available: https://radimrehurek.com/gensim/auto_examples/tutorials/run_doc2vec_lee.html