codenavi, a deep learning framework for identifying software bugs

Jeff Li, UC Berkeley, lije@microsoft.com

December 2021

Abstract: To facilitate better software development processes in projects with large groups of contributors and established commit histories, I propose a machine learning framework, codenavi, which utilizes deep learning language models to help developers identify software bugs. This framework implements a generalizable code parser across multiple languages and the application of neural machine language translation (NMT) models in order to identify the likelihood of a new bug associated with each new update, given the pre-existing history of the project. This framework is intended to aid developers in reviewing multiple code changes and accelerate developer velocity on complex projects. This paper covers the problem space, and a sample implementation for an open source project, matplotlib. This code for this project is publicly available and can be found at https://github.com/superli3/codenavi

Intro:

In large established software development projects, bugs can be extremely costly. An IBM study from 2010 estimated that bugs can be 5x more costly to identify & fix after it is deployed, compared to being identified in the design phase (Maurice Dawson, 2010). The software development process itself can be fast moving, complicated, and confusing, making bugs inevitable. Technical details are often siloed with certain developers, and information on project details may be lost over time as developers change projects. Fundamentally, buggy software often leads to a poor user experience and hurts a product or service's reputation.

The confluence of these trends in the software industry has given rise to practices and techniques to reduce the risk of shipping bugs. Some common industry practices involve structured processes such as continuous integration & continuous deployment (CI/CD), and practices such as test driven development. A common theme amongst many of these practices is that structure and process flow are critical for reducing the risk of shipping bugs for a project.

Existing literature

While the application of these processes has been applied in the software industry for many years, the application of natural language models in software development and directly to code is still in its infancy. However, over the last 3 years, there have been several papers on applying deep learning techniques to code. One notable paper in the domain was written by Uri Alon, Shaked Brody, et al, entitled Code2Seq, which leverages the structure of code, namely the abstract syntax trees (ASTs) to perform summarization and classification tasks (Raychev, 2016). More recently, a paper was written called "Evaluating Large Language Models Trained on Code", which utilizes a Generative Pretrained Transformer (GPT) language model tuned on publicly available code on GitHub, to create a seq2seq based translation model from docstrings to code (Mark Chen, 2021).

When turning an eye to the topic of applying neural network models to the area of bug detection in particular, the topic is not as well explored, and existing libraries and frameworks are sparse. The most notable paper in this domain is the paper DeepBugs, which uses name-based bug detection to identify bugs in snippets of code (Michael Pradel K. S., 2018). However, broadly speaking, the application of language models to bug detection is in a nascent stage, and typically is not deployed at scale.

Datasets

A key topic which should be discussed when delving into this topic is datasets and data quality. Existing papers up to this point published on this topic have focused on curated datasets, providing high data quality but leaving the overall scalability of these approaches relatively low. For example, Pradel and Sen utilized a dataset called Javascript150k (Raychev, 2016), which is openly available dataset. The authors of Code2Seq have a more "real world" approach, where they take a collection of top 100 starred Java projects on Github and train a seq2seq model

on the curated dataset. However, this approach did not leave much room for other languages, as the parser is language specific to Java. Furthermore, in the paper Software Analysis (Michael Pradel S. C., 2020), some of the proposed future contributions to this area involve a need for better data quality, as well as better ways to gather data. Generally speaking, there is a lack of common frameworks to generalize a deep learning based bug detector across multiple software projects. Data collection, quality, and processing are all challenges which limit how language models can be deployed.

codenavi

In this project, I look to create a generic, structured process to apply language models such as code2seq to the history of each software project. For most well-established large scale software projects, there will often be multiple contributors, and pull requests will often be utilized as the key mechanism by which code is often updated and reviewed by contributors in a project, prior to merging it with a main branch. Reviewers will often leave comments on code, which may go through multiple iterations. Given this software development process, I propose a framework by which pull requests are the key unit to look at for a NMT based classifier model.

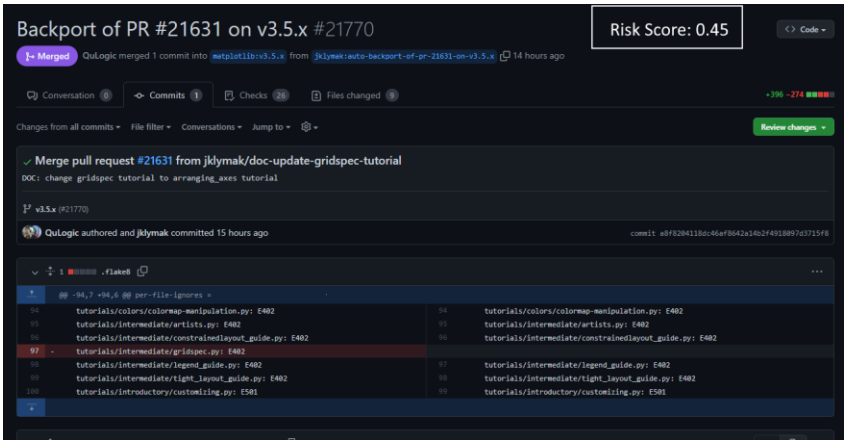


Figure 1: An example of how this information may be presented to the user

The goal here is to assist code reviewers by associating a "risk score" with each pull request. In an end state, the idea would be to present this score for each pull request. Presenting the information in this way allows for reviewing engineers to benefit from the model output, but still require human reviewer input and approval.

There are 4 main challenges when approaching this problem:

1. **Parsing:** A key element of turning machine level code into structured snippets is the parser that is used. Code is significantly different from human language. Not only do we need to be able to parse the code adequately, but ideally it can be done easily across multiple languages.
2. **Labeling:** A canonical problem when it comes to any ML based problem is how well you can label the dataset. Human input, the forum in which such issues are discussed, automation, and label accuracy are all key problems here.
3. **Compute/individual resources:** Parsing the files with one pull request can generate a large volume of data and may require large amounts of compute resources. The scalability limits here in regard to compute and storage significantly affects modeling as well, which will be discussed more in section 5 of methodology.
4. **Adequate models**: We want models that can accurately predict the likelihood of a bug associated with each new pull request. While I have designed the existing pipeline around the Code2Seq model, it is likely that this will be supplanted by a better language model down the line.

Evaluation

This problem as described is classification problem - is the pull request of interest or not of interest. I will be measuring the success of this approach using the metrics Precision, Recall, and F1 score. The training and evaluation process revolve around performing a 60/20/20 split between train/test/validation, and the efficacy of the model will be judged by how much better it performs compared to randomly picking a class (50%). For each labeled PR marked as “positive” in the dataset, I will be including one “negative” example in the dataset, to ensure as balanced a dataset as possible. While the end goal is to provide a risk score for the developer, these are present in the output from the softmax layer from the neural net decoder. As is, the F1/Precision/Recall currently serves as the key benchmark for this project.

Methods

For this project, the following methodology was employed.

1. Identify the project in question

While there are over 200 million different projects that are currently hosted on Github, there are several characteristics that one should put under consideration before trying to replicate this approach. For the purposes of this paper, I identified Matplotlib as a suitable project which had several key attributes. These characteristics are:

- **An extensive history of pull requests commits**: a project should have a suitable history of Pull Requests that can be pulled. Matplotlib in particular is an open source library which is commonly used for visualization with Python, and has a rich history of pull requests (13,220 as of writing).
- **A history of bugs that can be examined**: applying project level bug detection for a project requires a history of bugs that can be labeled, either manually or via automated processes. This is covered more in depth in the labels section.
- **A codebase that can be sufficiently parsed into a format which can be understood by a language model**: Once the history for a particular project has been pulled, we want to make sure that the underlying code could be analyzed as part of a language model. The decision on which code language model to use is discussed more extensively below. I had identified code2seq as promising candidate for a model, as it had previously shown promise compared to other language models for understanding code. However, code2seq this required the ability to parse the code down into the proper format. This came with multiple challenges, which will be covered in section 3 below.

2. Scrape a PR history of the project

Data was scraped using the GitHub Pull Request API. Several important fields were pulled to facilitate the data processing:

- The commit hash associated with the pull request ID (e.g., 25ff2ed)
- The corresponding .diff file, which serves as a manifest of which files were changed as part of the pull request. An example is provided in the appendix.
- Isolate the files that were changed as part of each merge request from the .diff file, and which can be parsed (e.g., .py files)

3. Preprocess data for model

A discussion on model selection and data preprocessing

The underlying method to convert code to a format which can be read by neural nets models is still an area of active research. It is suggested in the code2seq paper that converting code to AST based sequences, which up to 2021 seemed to have the most promising results when it came to code summarization and classification. However recent advancements, namely OpenAI Codex suggest that applying large language models directly to code may have success – Github's Copilot has had very impressive results when it comes to seq2seq machine translation. However, the ability to repurpose Codex for code classification based on custom labels is limited as of writing. Due to limited access to repurpose OpenAI Codex for bug classification, I proceeded with parsing the code via AST based approach. However, advancements down the line may allow codenavi to work with Codex.

Regardless, the underlying method should be carefully considered, as the process of converting code to a format which can be understood by a language model is nontrivial and involves significant engineering effort. The code processing process for codenavi using a code2seq model is highlighted below.

For qualifying files, generate AST paths, and traverse paths into sequences

For prior implementations of applying language models to parsed ASTs, researchers have typically used code that has been largely preprocessed already. In the cases where researchers wrote parsers, they were typically language specific to one language. However, this may not generalize well to a real world software project, as it is common to have a project that may leverage multiple languages at once. In Matplotlib's case, this covers both Javascript and Python. Furthermore, any code that cannot be parsed will escape any evaluation.

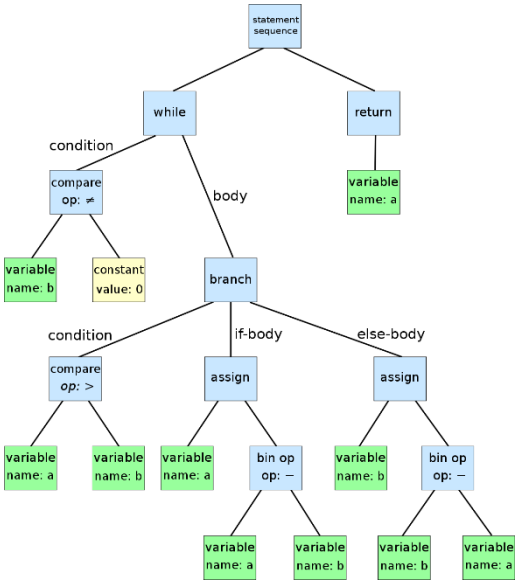


Figure 2: An example of an AST based on a snippet of code. Source: Wikipedia

In order to create a generalizable framework, the project should ideally cover multiple languages. After extensive trial and error, I utilized the library semantic, which was developed by Github (Github) and converted the code into AST based representations in json format. This implementation allows for all languages supported by semantic to be leveraged for a language model – over 10+ languages, including but not limited to, Python, Javascript, Java, Go, etc. An example of the parsed .json tree can be found in the Appendix – Parsed tree snippet.

After the data was converted into AST based representations, I wrote a custom tree traversal program to convert the json data structure output by semantic into sequences that could be utilized by the Code2seq model. I created a parser here to convert the AST based json structure into sequences. These paths are subsequently concatenated together to create a structured representation of a function or class.

Fundamentally, given the AST of a code snippet, with j paths and a max tree depth of k, the paths can be represented as $(v_1^1 v_2^1 \dots v_k^1) \dots (v_1^j v_2^j \dots v_k^j)$. As the depth of the tree parse increases, the size of the dataset grows exponentially.

An example of generated sequences can be found in the Appendix – Code sequence.

4. Labels

Labeling is a critical portion of any machine learning project. For this project, I went through the history of the comments in the Matplotlib and manually labeled resolved issues that could be tied back to pull requests or commits, based on the comments of the contributors. The Matplotlib contributors are proactive in their efforts to paste links to pull requests and issues, which made labeling for this project relatively straightforward. While not all threads mentioned a particular pull request or commit, I was able to label 180 pull requests as "problematic" out of 6000+ issue threads. Issues without a label were assumed to be of the negative class. For future implementations, it is possible that such efforts can be automated to a high degree, depending on the forum in which such bugs are discussed.

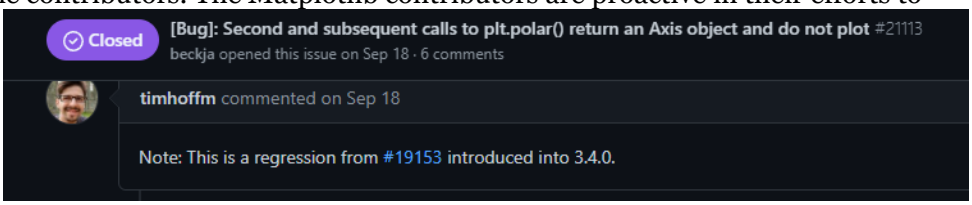


Figure 3: An example of a specific pull request mentioned in a discussion of an issue

5. Storage, dataset construction, splits

Storage

Parsing the data by converting the data to AST representations and subsequent sequences resulted in the underlying dataset growing exponentially. Notably, creating these AST based representations for all qualifying files across all historical pull requests was extremely time intensive and resulted in a huge explosion in data size. For reference, the original pulled files from the matplotlib repo, after only pulling changed files, yielded a dataset of 1.5 GB. Parsing the

code down into AST based trees resulted in the dataset expanding by approximately 12 fold, close to 20GB. Furthermore, depending on how the AST based trees were parsed into sequences, the overall dataset size could expand by upwards of an additional 40 fold, well over a half a terabyte. The processing time was also considerable, taking multiple days to go over the entire code history of the project. Additional information on the infrastructure used to parsed the data can be found in the Appendix - Infrastructure.

Table 1: Data size across the codenavi's processing pipeline

Description/Step	Data Size
Original scraped data of only changed files, 12530 scraped commits	1.5 GB
Creating a balanced dataset of 307 commits, only files identified in diff	25 MB
Creating JSON trees from 307 commits	309 MB
Converting JSON trees to sequences, depth 2	33 MB
Converting JSON trees to sequences, depth 4	656 MB
Converting JSON trees to sequences, depth 8	14GB

Dataset construction, splits

In addition to storage and processing time, labels were a significant factor when constructing the dataset for training. I experimented with several positive/negative splits during the modeling phase. Implementing a 1:1 balance between positive and negative labels tended to yield the best results. Other splits such as a 3:1 or 5:1 split between label classes often resulted in the model predicting only one class – leading to high, but misleading F1 scores.

Proceeding with a 1:1 positive/negative training set shrunk the dataset down considerably. Only 307 of the given 12530 repos were leveraged as part of this project. Even so, the largest version of the parsed dataset yielded a total dataset size of 14GB. Table 1 illustrates the exploding nature of data given the approach.

6. Modeling

After running the data processing pipeline and setting up the requisite infrastructure, I set on running models on the given dataset. A shallow parse depth of 2 served as the initial baseline for rapid iteration, to discover the best parameters for training. Additional training was done on generated sequence datasets of deeper parse depths of 4 and 8. I opted for a 60/20/20 train/test/validation split when training the pipeline.

Model	F1 Score
Code2seq, parsed depth 2	0.533
Code2seq, parsed depth 4	0.52
Code2seq, parsed depth 8	0.533

Table 2: F1 Scores

Code2Seq

Code2seq uses a standard encoder-decoder architecture for NMT. Instead of reading the input as a flat sequence of tokens, the encoder creates a vector representation for each AST path (denoted as z).

Notably, as the code2seq model here is being adopted for 1 word length classification, it should be noted that the initial state of the model is generated by averaging the combined representations of all k paths, and z serves as a vector representation of the path. Additional details on the vector representation can be found in Appendix – Vector representation.

The initial hidden state of the decoder, which is used for classification, is then generated by averaging all of the combined representations of k paths.

$$h_0 = \frac{1}{k}\sum_{i=1}^{k} z_i$$

Modifications for codenavi

For the purposes of codenavi, I converted the beginning of each sequence the represent the labeled class, and then modified the code2seq configuration to output a one length sequence. Additionally, the code2seq was repurposed to train for up to 300 epochs to look for the optimal precision. Otherwise, despite several rounds of experimentation, the default settings for code2seq were used when retraining the model.

Overall, the baseline model scores of codenavi suggests that there is a slight uptick over running from random when it comes to running a language model. Generally speaking, the performance appears to be slightly better than flipping a coin at random. However, additional traversal of the trees yielded little to no benefit for the F1 score. While the initial results here show little to no benefit over a shallow parse, there are several areas of improvement for this approach that may be leveraged down the line.

Error Analysis, Areas of Improvement

Given this application of code2seq for codenavi, there are several areas for improvement here. While none of these aspects prohibited the creation of an end to end machine learning pipeline here, it is likely that improvement on these aspects would aid in improving the F1 scores.

- **Missing commit hashes, no code:** When scraping the data from Github, there were several hundred pull requests that were missing a corresponding commit. In addition, there were a few instances where some pull requests did not contain any code that could be parsed. This may occur, when the pull request addressed a simple change, e.g. a configuration change in a .yaml file, or the file changed was in regards to something such as documentation.
- **Unable to parse code to AST based sequences:** There were issues which came to parsing the underlying qualifying files to .json. Approximately 10% of the qualifying python files could not be parsed by the semantic library, which also affected the limited set of positive labels that were available. In addition, there is additional room to improve the parser and data processing framework for codenavi. For instance, codenavi currently only supports parsing code starting from class objects. If a script does not have a class object in the tree, no tree object will be returned, even if there was code changed in the script.
- **Getting granular in identifying changes:** Parsing could also be improved to better isolate the changes that were made when parsing into sequences. Currently, the changes are only identified on a file by file level. However, it is plausible that a PR may only change a few lines out of a thousand line file, adding additional noise. Typically, within a pull request, only the highlighted changes will be covered and shown to the user in the .diff file (see Appendix – Pull Request, .diff). Improving the parsing may improve the "signal" of the functions or classes that are changed as part of a pull request. This is also supported by the authors of code2seq, who mention that short code snippets (i.e., less than 3 lines), tend to perform better compared to longer code snippets.
- **Better models:** As approaches to parse code continue to improve, it is entirely likely that better classification approaches for code will emerge in the future. Codenavi has been written in such a way that the underlying scraped code can be utilized for other language models down the line.

Conclusion

As the reliance on software in the world goes, software quality is paramount to ensuring good experiences for customers and developers. Codenavi is a framework to take advantage of recent advancements in using neural machine translation models for classification. The initial results of codenavi applied to the development history of matplotlib paint a mixed results for applying natural language models to the software development process. However, recent advances in the natural language space, namely frameworks like OpenAI Codex open the door forward to rapid enhancements in the future.

Acknowledgements

I would like to thank my instructor at the UC Berkeley MIDS program, Joachim Rahmfeld, as well as my colleagues at Microsoft (Divya Gorantla, Hua Ding, Vasanth Ramani, Yingying Chen and others) for providing feedback along the way.

References

Github. (n.d.). Semantic. Retrieved from https://github.com/github/semantic

Mark Chen, J. T. (2021). Evaluating Large Language Models Trained on Code. *arXiv.org*. Retrieved from https://arxiv.org/abs/2107.03374

Maurice Dawson, D. B. (2010). Integrating Software Assurance into the Software Development Life Cycle (SDLC). *Journal of Information Systems Techonlogy & Planning*, 49-53. Retrieved from https://www.researchgate.net/publication/255965523_Integrating_Software_Assurance_into_the_Software_Development_Life_Cycle_SDLC

Michael Pradel, K. S. (2018). DeepBugs: A Learning Approach to Name-based Bug Detection. *arXiv.org*. Retrieved from https://arxiv.org/abs/1805.11683

Michael Pradel, S. C. (2020). Neural Software Analysis. *arXiv.org*. Retrieved from https://arxiv.org/abs/2011.07986

Raychev, V. B. (2016). Learning Programs from Noisy Data. *n Proceedings of the 43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Retrieved from https://www.sri.inf.ethz.ch/js150

Appendix

Parsed Tree Snippet

Please note that only a small snippet is provided below, for brevity.

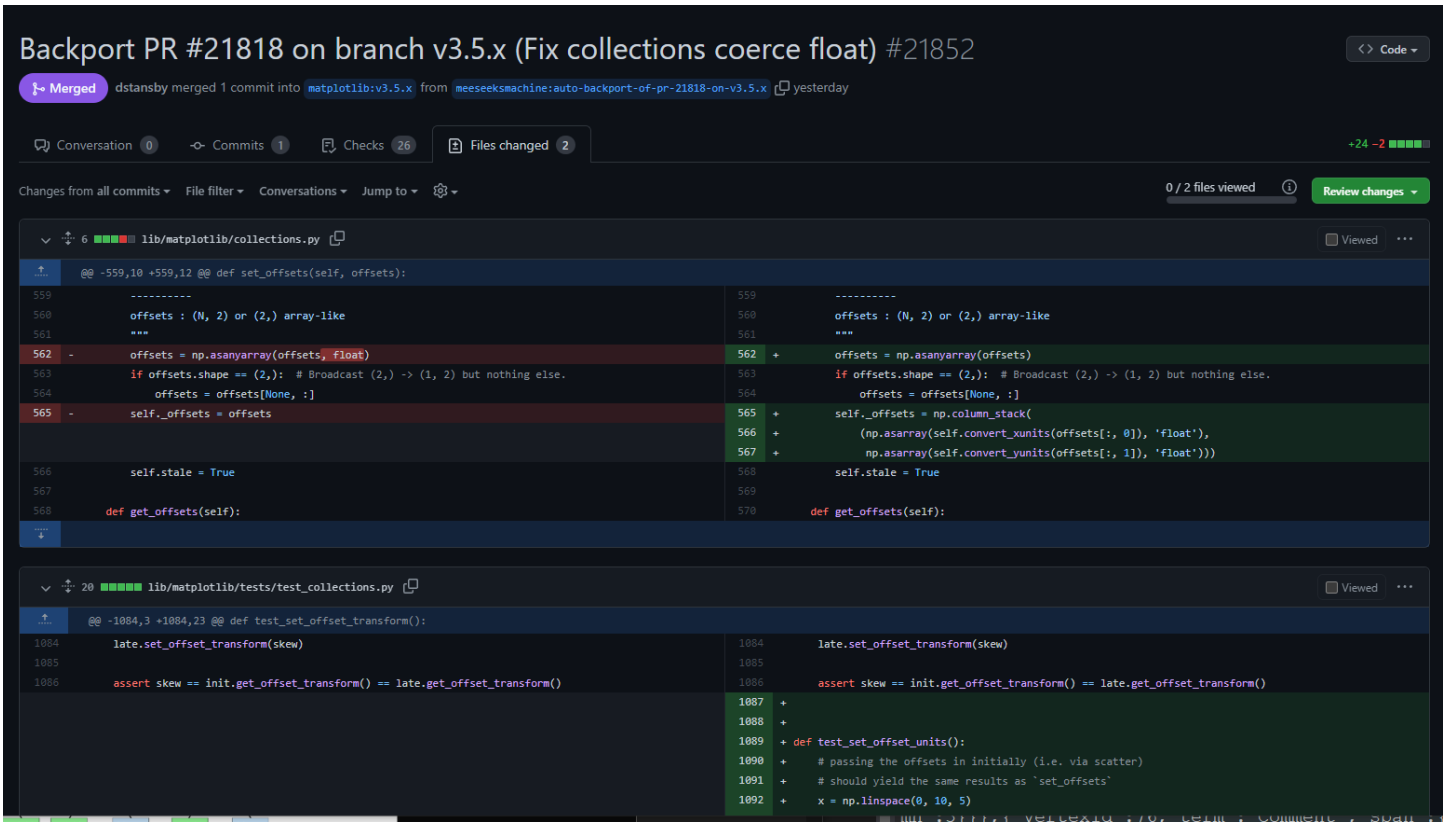

```
{"files":[{"path":"/home/output/0a4fc953673f8799e18440c74237d8c2a4848b59/0a4fc953673f8799e18440c74237d8c2a4848b59_customizing.py","language":"Python","vertices":[{"vertexId":1,"term":"Statements","span":{"start":{"line":1,"column":1},"end":{"line":195,"column":1}}},{"vertexId":2,"term":"TextElement","span":{"start":{"line":1,"column":1},"end":{"line":19,"column":4}}},{"vertexId":3,"term":"QualifiedAliasedImport","span":{"start":{"line":21,"column":8},"end":{"line":21,"column":19}}},{"vertexId":4,"term":"Identifier","span":{"start":{"line":21,"column":17},"end":{"line":21,"column":19}}},{"vertexId":5,"term":"QualifiedAliasedImport","span":{"start":{"line":22,"column":8},"end":{"line":22,"column":32}}},{"vertexId":6,"term":"Identifier","span":{"start":{"line":22,"column":29},"end":{"line":22,"column":32}}},{"vertexId":7,"term":"QualifiedAliasedImport","span":{"start":{"line":23,"column":8},"end":{"line":23,"column":25}}},{"vertexId":8,"term":"Identifier","span":{"start":{"line":23,"column":22},"end":{"line":23,"column":25}}},{"vertexId":9,"term":"Import","span":{"start":{"line":24,"column":1},"end":{"line":24,"column":26}}},{"vertexId":10,"term":"Alias","span":{"start":{"line":24,"column":20},"end":{"line":24,"column":26}}},{"vertexId":11,"term":"Identifier","span":{"start":{"line":24,"column":20},"end":{"line":24,"column":26}}},{"vertexId":12,"term":"Identifier","span":{"start":{"line":24,"column":20},"end":{"line":24,"column":26}}},{"vertexId":13,"term":"Call","span":{"start":{"line":25,"column":1},"end":{"line":25,"column":24}}},{"vertexId":14,"term":"MemberAccess","span":{"start":{"line":25,"column":1},"end":{"line":25,"column":14}}},{"vertexId":15,"term":"MemberAccess","span":{"start":{"line":25,"column":1},"end":{"line":25,"column":10}}},{"vertexId":16,"term":"Identifier","span":{"start":{"line":25,"column":1},"end":{"line":25,"column":4}}},{"vertexId":17,"term":"Identifier","span":{"start":{"line":25,"column":5},"end":{"line":25,"column":10}}},{"vertexId":18,"term":"Identifier","span":{"start":{"line":25,"column":11},"end":{"line":25,"column":14}}},{"vertexId":19,"term":"TextElement","span":{"start":{"line":25,"column":15},"end":{"line":25,"column":23}}},{"vertexId":20,"term":"Empty","span":{"start":{"line":25,"column":24},"end":{"line":25,"column":24}}},{"vertexId":21,"term":"Assignment","span":{"start":{"line":26,"column":1},"end":{"line":26,"column":27}}}
```


Code Sequence

Please note that only a small snippet is provided below, for brevity. Note that the beginning word “o” denotes the negative class.

```
0 class METHODNAME,Class|Identifier,identifier METHODNAME,Class|Statements,statements
statements,Statements|TextElement,text|element
statements,Statements|Assignment,assignment ….
```

Pull Request - .diff



While the code reviewers may often see the code via the interface above, git uses the .diff files to show which lines of code were changed. An example of this raw .diff file is provided below.

```
diff --git a/lib/matplotlib/collections.py b/lib/matplotlib/collections.py
index 562362ab0c3..1a20e66e023 100644
--- a/lib/matplotlib/collections.py
+++ b/lib/matplotlib/collections.py
@@ -559,10 +559,12 @@ def set_offsets(self, offsets):
          ----------
          offsets : (N, 2) or (2,) array-like
          """
-         offsets = np.asanyarray(offsets, float)
+         offsets = np.asanyarray(offsets)
          if offsets.shape == (2,):  # Broadcast (2,) -> (1, 2) but nothing else.
              offsets = offsets[None, :]
-         self._offsets = offsets
+         self._offsets = np.column_stack(
+             (np.asarray(self.convert_xunits(offsets[:, 0]), 'float'),
+              np.asarray(self.convert_yunits(offsets[:, 1]), 'float')))
          self.stale = True

      def get_offsets(self):
diff --git a/lib/matplotlib/tests/test_collections.py b/lib/matplotlib/tests/test_collections.py
index 9f92206e51c..bc837f8db51 100644
--- a/lib/matplotlib/tests/test_collections.py
+++ b/lib/matplotlib/tests/test_collections.py
@@ -1084,3 +1084,23 @@ def test_set_offset_transform():
      late.set_offset_transform(skew)

      assert skew == init.get_offset_transform() == late.get_offset_transform()
+
```

```
+
+def test_set_offset_units():
+    # passing the offsets in initially (i.e. via scatter)
+    # should yield the same results as `set_offsets`
+    x = np.linspace(0, 10, 5)
+    y = np.sin(x)
+    d = x * np.timedelta64(24, 'h') + np.datetime64('2021-11-29')
+
+    sc = plt.scatter(d, y)
+    off0 = sc.get_offsets()
+    sc.set_offsets(list(zip(d, y)))
+    np.testing.assert_allclose(off0, sc.get_offsets())
+
+    # try the other way around
+    fig, ax = plt.subplots()
+    sc = ax.scatter(y, d)
+    off0 = sc.get_offsets()
+    sc.set_offsets(list(zip(y, d)))
+    np.testing.assert_allclose(off0, sc.get_offsets())
```

Vector Representation

This portion covers the vector representation that is utilized from the code2seq model and is provided as a reference for the reader's convenience. For additional information, please reference the code2seq paper.

Fundamentally, the vector representation, *z*, is a combined representation from both paths and tokens. Given the limited vocabulary of the parse tree, the vocabulary is represented using a learned embedding matrix E^{nodes}.

The path representation is generated as follows, where the encode path is generated using the final states of a bidirectional LSTM.

$$h_1, \ldots\ldots, h_l = LSTM\left(E_{v_1}^{nodes}, \ldots E_{v_l}^{nodes}\right)$$

$$encodepath(v_1 \ldots. v_l) = [\overrightarrow{h_l}; \overleftarrow{h_l}]$$

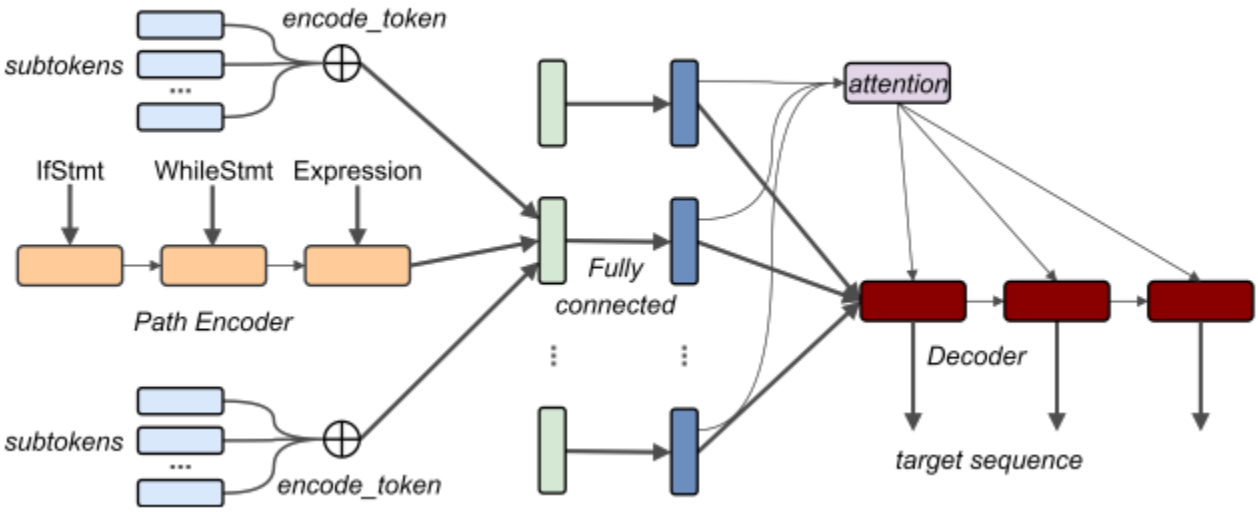
The token representation is as follows, generated from the terminal nodes of the path. Subtokens are split where variables such that a token like *ArrayList* would be split into the sub tokens *Array* and *List*.

$$encodetoken(w) = \sum_{s \in split(w)} E_s^{subtokens}$$

The combined representation z is generated as follows

$$z = tanh\left(W_{in}\left[encodepath(value(v_1 \ldots vl)); encodetoken(value(v_1)); encodetoken\left(value(v_l)\right)\right]\right)$$

Where W_{in}represents a hidden matrix.



Infrastructure

The initial scraping and preprocessing were done locally on a mid-high range Windows 10 desktop (Intel 10900X processor, 64GB of RAM), within a Python virtual environment or Docker containers. For the modeling portion of the pipeline, the data was copied into an Amazon S3 bucket, and trained on a Nvidia Deep Learning AMI instance on AWS – using the g4dn.xlarge instance, with a Nvidia T4 GPU. Once the VM was provisioned, modeling was done within Tensorflow container from Nvidia's NGC catalog, using guidance from the existing code2seq library.