Topological Data Analysis on Source Code Embeddings

Background and Motivation

Topological data analysis (TDA) is an emerging approach for analyzing complex datasets by extracting shape-based features and structural insights. While TDA has been applied to various domains, its use in studying source code is a relatively new and developing area. Still, it shows promise for providing new insights into code structure, quality, and evolution.

Source code repositories contain vast amounts of complex, high-dimensional data that can be challenging to analyze using traditional methods. TDA offers a unique perspective by focusing on the topological and geometric properties of data, which can reveal hidden structures and patterns.

One of the foundational elements of TDA is the concept of persistence diagrams, which shows the lifetimes, or the "persistence," of topological features in a dataset as certain parameters change over time. As an example, imagine you're looking at a mountain range from far away. As you get closer, you start seeing different *features*: First, you might see one big mountain. Then as you get closer, you notice there are actually several peaks. Even closer, you might spot valleys and caves. And when you move past them and start moving farther away, the valleys and caves disappear, then the peaks, and finally the mountain. Here, the mountain "persists" the longest, and the valleys and caves the shortest.

In the context of topology, data points, when plotted on a point cloud, form shapes that have features such as the number of clusters that form a "shape" and the number holes in the shapes. Topology analyzes how these features "persist" in what we call persistent homology.

Persistence diagrams summarize the topological features of a dataset across multiple scales, and they provide a compact representation of the data's shape, allowing for the identification of significant features that persist across various "resolutions" (Silva et al., 2017; Bubenik & Hartsock, 2022). In the context of source code, persistence diagrams can be utilized to analyze

the relationships between different components of the codebase, such as classes, methods, and their interactions. This can help in identifying modular structures (i.e., the independent subcomponents of a program's functions) and potential areas for refactoring (i.e., improving internal code without changing the external functionalities) (Batista et al., 2018).

Data Description

CodeEmbeddingsGenerator.py reads Java code files, then uses the code to generate code embeddings through a systematic method:

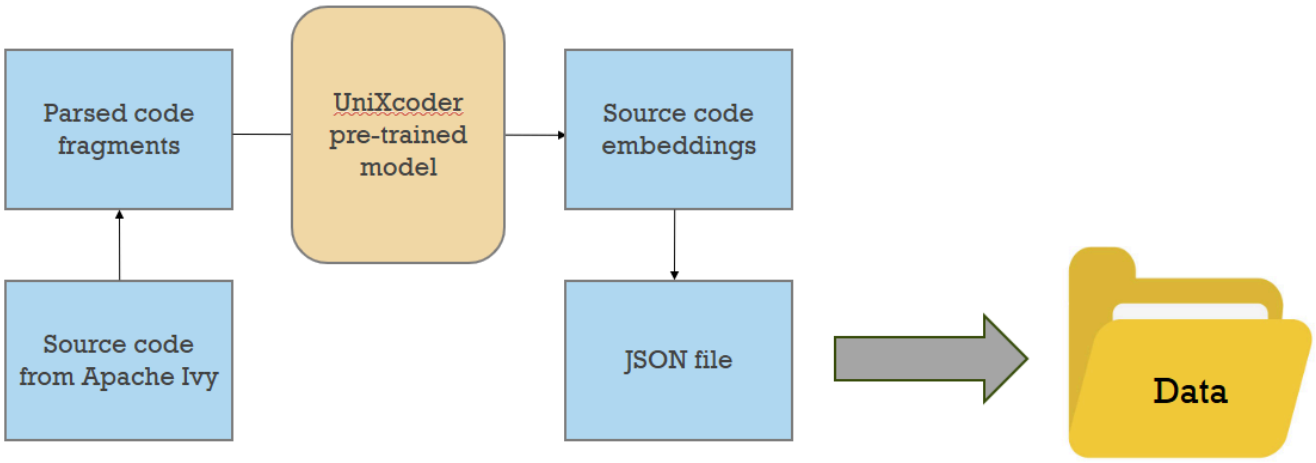
1. Extract code fragments using the Tree-sitter parser at three levels: classes, methods, and tokens. The classes are the largest code fragments, which contains the smaller method code fragments, which contains the even smaller token code fragments.

2. Input the code fragments into the UniXcoder neural network model for AI code generation, which outputs code fragment embedding representations. Each embedding includes 768 values.

3. Save each embedding as a JSON file in a data folder, along with some other relevant data.

    - Includes the class/method name if it is a class/method embedding, or the token itself if it is a token embedding.
    - Includes the string indices for parsing out the code fragment being represented.

Here is an example data engineering pipeline, where the source code is taken from the official Apache Ivy GitHub repository:



Furthermore, the directory in the data folder is structured similarly to the structure of the source repository. JSON files that were created from a certain Java file would be stored in a folder named after that same Java file, which is stored in a location that mimics that of its source. JSON files are also named after the classes or methods they are in, including those that are nested. The idea is to simplify the process of separating embeddings by granularity and directory locations for ease of analysis.

Here are some example JSON filenames that were generated from JarModule.java (from Apache Ivy):

Filename	Meaning
c.JarModule.json	Class embedding of JarModule class
c.JarModule_m.getJar.json	Method embedding of getJar method found in JarModule class
c.JarModule_m.getJar_token.json	Token embedding of a token found in getJar method and JarModule class
c.JarModule_m.getJar_token(0).json	Token embedding of a different token found in getJar method and JarModule class
c.JarModule_m.getJar_token(1).json	Token embedding of yet another token found in getJar method and JarModule class
token.json	Token embedding of a token not found in any class or module

Progress and Next Steps

Now that we have obtained our embeddings, our next step is to place them into their own JSON files, then store them in a brand-new directory in a way that reflects the directory in which the original source files are organized. This will preserve information on the original code structure, which we will use in our data analysis.

As long-term goals, we aim to (1) conduct exploratory data analysis on embedding point clouds, (2) compute topological features of the embedding point clouds, and (3) analyze their persistent homology.

References

[1] Batista, N., Sousa, G., Brandão, M., Silva, A., & Moro, M. (2018). Tie strength metrics to rank pairs of developers from github. Journal of Information and Data Management, 9(1), 69. https://doi.org/10.5753/jidm.2018.1637