Code Retrieval based on Weighted Similarities of Control Flow and Structure

Suraj Pandey
M Tech CSE
IIIT Delhi

William Scott Paka
M Tech CSE
IIIT Delhi

Udit Pant
M Tech CSE
IIIT Delhi

Abstract

This project aims to find relevant answers for an input query consisting of the problem description and its associated code. It is often a common practice among software developers to look up errors on the web. This process is time-consuming and in cases when the query is not well-formulated, can result in irrelevant search results. By creating a system which fetches top answer-code pairs for the input query, our attempt is to bridge the gap between the developer and relevant search results. In this paper we provide combination of different similarity techniques for matching text and code. We use tf-idf based measures for text similarity and control-flow measures for code similarity. We use data collected from StackOverflow with questions related to Python. Evaluation on test set yielded a Mean Average Precision (MAP) of 0.281.

1 Introduction

Software developers frequently perform searches on the web related to errors or bugs in code. For a fruitful search, it is important to formulate the query efficiently. Web searches often return irrelevant results due to improper matching of the input query. The entire process of correctly formulating the query, searching and repeating the process until we get the right results is time-consuming. The problem of code retrieval is to identify similar pieces of code with similarity in the domains of both syntax and semantics.

Code similarity, unlike text similarity, is one of the non-trivial aspects of Information Retrieval paradigm. What differentiates code retrieval from other retrieval models is its sophistication in terms of structural and semantic complexity (Ponzanelli et al., 2013). Most traditional retrieval models applied on problems rely on text-based similarity. However, this is not the case when it comes to code-based similarity. There exist little or no notion of text similarity when it comes to code retrieval due to the dominating presence of variables as the variables of the programs are named at the convenience of the programmer. However, code sections consist of keywords in large numbers which can be useful in understanding the structure of the code (Ponzanelli et al., 2014). The task of generalizing code-level similarity over languages is difficult due to the varying syntactic structure despite the use of similar conditional-flow constructs. To understand the code at a semantic level, we need to apply non-trivial techniques to be able to compare and evaluate the degree of similarities (Bacchelli et al., 2012).

In this paper we aim at developing a robust code retrieval model that is capable of fetching relevant code snippets for the input query while ensuring a good measure of precision and recall. We implemented a model to convert the code into its control flow graph considering all the possible cases of flow change in python and have used similarity techniques based on eigen vectors, isomorphic structure of graphs and steady state of the adjacency matrices. We have also used different weighting factors to get the best possible combinations of the graph and text similarity.

2 Related Work

Most of the previous works on code retrieval focus on developing plugins for IDEs in order to reduce the costs associated with manual searching on the web (Ponzanelli et al., 2014). Zagalsky et al. (Zagalsky et al., 2012) have developed a live system - Example Overflow which makes use of social media for code recommendation. Their methodology follows a keyword search based approach implemented through *tf-idf* weighting using Apache Lucene library. To build the search index, both

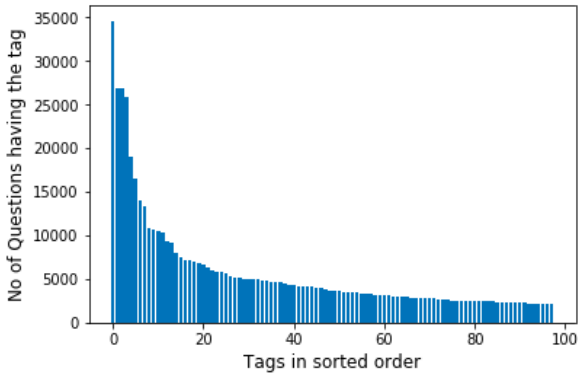

Figure 1: Plot of tags vs occurrence of tags in questions

code snippet and related metadata is used.

Yin et al. propose a novel way to extract aligned code and natural language pairs (Yin et al., 2018). In their model task of retrieving code has been modelled into a classification problem determining similarity among graph nodes by neighbor matching. Nikolic et. al. (Nikolić, 2012) proposed a method to calculate similarity among nodes by averaging the similarities together between the neighbours at both ends of edges. Allaman et. al. (Allamanis and Sutton, 2013) presented a topic modelling that combines question concepts with codes. Few other works include usage of neural networks to summarize the code into understandable format (Iyer et al., 2016).

3 Dataset

The dataset used is a collection of Python questions from Stack Overflow and can be found at `https://tinyurl.com/y41846hl`. It is organized into three csv files consisting of Questions, Answers and Tags. Questions contain attributes such as Id, OwnerUserId, CreationDate, Score , Title and Body. Answers contain attributes such as Id, OwnerUserId, CreationDate, ParentId (corresponding question id), Score and Body. Tags contain pairs of QuestionId and Tag.

3.1 Analysis

Statistics of the dataset are as follows: Questions - 6 lakhs, Answers - 9.8 lakhs, Tags - 18.8 lakhs. The total unique Tags are 16,896 and top 5 tags make up of 40% of the Question-Tag pairs. By default, code is embedded in the body parts of Questions and Answers enclosed in ¡code¿ xml tag. There could be multiple codes in both questions and answers. Figure 1 shows the bar plot of the tags and their occurrence in the questions in sorted order.

4 Methodology

4.1 Preprocessing

4.2 Text Preprocessing

As we have the code embedded into text, it is separated from the text part for effective usage. The text parts are then preprocessed using techniques such as case folding, punctuation removal, stop words removal and lemmatization.

4.2.1 Code Sampling

Each question in the dataset has multiple answers corresponding to it. However, all these answers are not relevant to our model due to the presence of code which are not in python. Such code snippets are removed from the dataset by employing a searching technique that looks for the presence of keywords of python language. The search is successful when the code snippet contains python keywords greater than a fixed threshold or else the code is discarded. Following this process, we sample the questions from the entire dataset to reduce the search time and also to increase the precision.

4.3 Data Representation

4.3.1 Text

Text from title and body sections is represented in the form of vectors. We use TF and TF-IDF techniques to convert text to vector format. Similarity among vectors is computed based on both the distance metric and cosine similarity.

4.3.2 Code

For representing code, control flow of the entire chunk of code is identified. This flow is best captured with the help of control flow graphs (CFG). The benefit of using CFG for code representation is that we can generalize codes based on their control flow. However, any form of computation is difficult to be carried out using graphs solely. Therefore, these graphs are further transformed into adjacency matrices which provides for easy mathematical calculations to identify similarity between matrices and henceforth similarity in codes

The algorithm is structured to work in three passes, narrowing down our search space in each pass aiming for the relevant answer for the query.

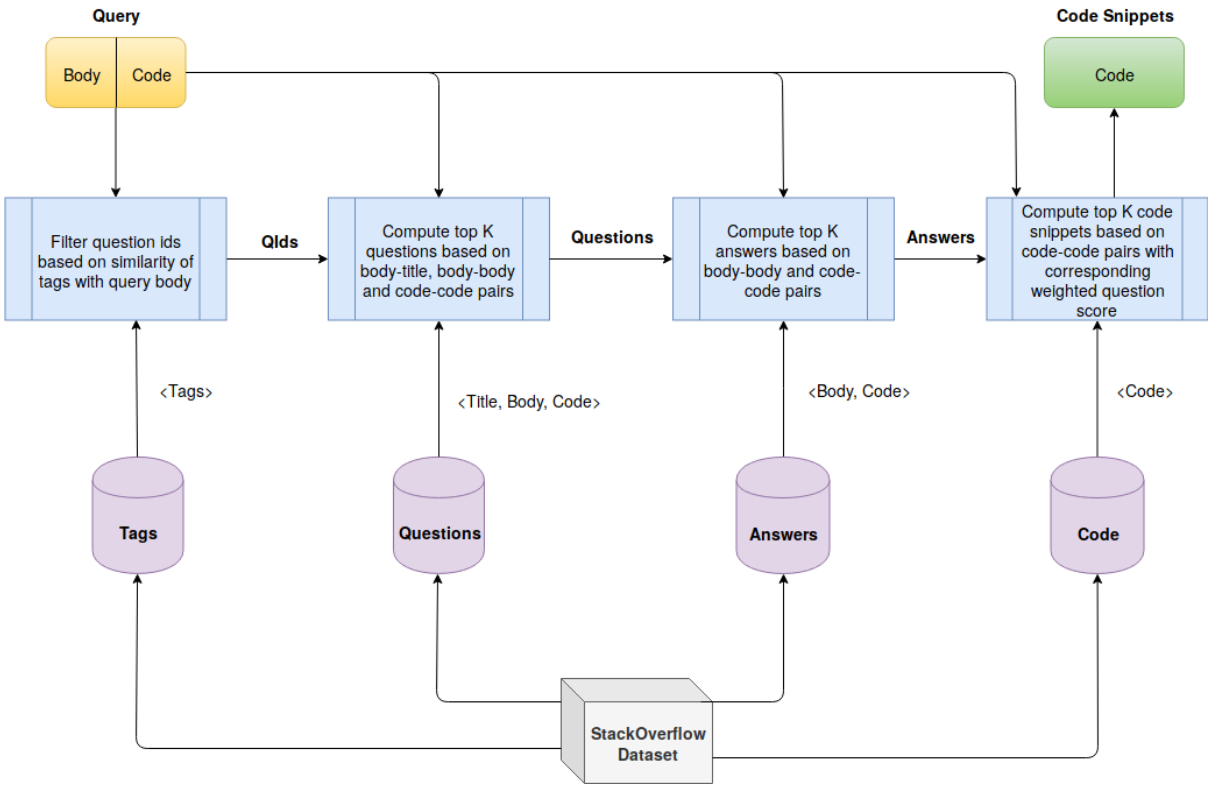

Figure 2: Model Workflow

It is designed to take query as question code pair as input and in the end return a set of proposed answers.

The three passes in our model are as follows:

1. Finding relevant Tags
2. Finding relevant Questions
3. Finding relevant Answers

4.4 Finding Relevant Tags

This is an important stage of our model as in this stage we reduce the search space considerably, saving us a lot of computation which in turn helps our model to compute faster. As our dataset has a vast number of questions in which each question contains a title, body and code section and comparing the query with all of the available questions is not efficient. As a countermeasure, we utilize the tag fields corresponding with every question by taking the query words present in the tags to sample the questions. And in cases where no word matches the tags, we run on the whole questions.

4.5 Finding Relevant Questions

In this stage we filter the questions by finding the relevance of the query with all the sampled question from the above section. We make use of both title and body present in questions. The query title is matched with both the question title and question body using textual similarity mentioned in section 4.7. The code in the query is matched with code in questions using the similarity techniques mentioned in section 4.8. The final similarity measure is weighted as follows: query text and question title - 0.3, query text and question body - 0.3, query code to question code - 0.4. The total similarity will be in the range of 0-1. Top k (k=3) questions are filtered in this section and passed to the next.

4.6 Finding Relevant Answers

As the questions are sampled to k questions in the above section, we will now find the relevant answers from these k questions. The same similarity techniques used for the question sample above are utilised. As the answer do not contain title, the query text is matched with answer text and the query code is matched with the answer code. We have also used the question matching score that

was used to sample the questions as one of the weights. The final score of the answer is weighed as follows: query text and answer text:0.3, query code and answer code - 0.3, question matching score - 0.4. The range of the similarity will be in the range of 0-1. We display the top p (p=3) answers as the final solution.

4.7 Text Similarity

In the second phase, for matching query text with title and body sections, we represent title and body sections as two separate corpora. This is done by combining all titles together and all body sections together individually. A vocabulary is built for each corpus by picking out unique words. Each title and body section is then picked and transformed to vectors according to their respective corpus. The input query is also transformed to a document vector by considering the corpus against which it is being matched, i.e., title's corpus when similarity is being computed against titles and body's corpus when similarity is being computed against body. We then proceed to calculating tf-idf for each vector according to the corpus and represent it in the already-built vectors. These vectors are then used for calculating the cosine similarities. A similar procedure is followed for the third phase where only a single corpus is built for the body sections by combining all of the sections together.

4.8 Code Similarity

4.8.1 Sliding Window Syntax-Matching Heuristic

The sliding window heuristic compares two snippets of code by comparing n number of lines at a time where n is the smaller length out of the two code snippets. For each window-window comparison, each line of a window is matched with its counterpart. On completion, the siding window slides by one line and this process is repeated until we reach the end of the longer code snippet. Scores for each window-window comparison is stored and the maximum out of these scores is returned as the similarity score.

4.8.2 Control Flow

When measuring similarity in code-code pairs, one nuisance is the presence of large amount of variables in code. Since two snippets of code can have totally different variable names, two similar codes can be treated as different if this is not properly handled. To overcome this problem we are using control flow structure to represent code. Control flow is the order in which the statements get executed. Creating a control flow for the code will help compute a better similiarity measure. We considered the hash of each line in the code as a state. We analyzed the flow of different structures in python and created a control flow. The main structures we considered are if, if-else, if-elif-else, while, for, class, and def. Figure 3 shows the control flow of these structures. As python uses white space indentation to delimit blocks, we created a general function which can handle all of the indents. This general function can also handle the structures which were not manually constructed. The states are connected as depending on the flow in the structure, forming the edges. The edges are then converted to a adjacency matrix. As this is a real world dataset from stack overflow, the codes could be highly incomplete and we made the control flow generator robust enough to consider all the corner cases.

As described in the previous section, we take the control flow of the code as one of the representation. The control flow is then converted to its equivalent adjacency matrix for better mathematical computations. Five similarity measures that are mentioned below are computed for the adjacency matrix. The whole similarity measure is scaled from [0-1] with different weights assigned to different similarities.

- **Isomorphism:** Two graphs are said to be isomorphic if they have the same number of edges, vertices and if they are connected in an identical way. In our case, if two control flows are similar we consider that both the codes are a match we give complete similarity weight to it and if they are not isomorphic then we compute a similarity score based on a weighted average of the remaining four similarity measures described below.

- **Eigen Values Similarity:** Eigen vectors are the values that remain unchanged when a linear transformation is applied. Top eigen vectors denote the useful components, so we extract the **top k eigen values** for the matrices and computed the mean squared difference (Figure 4). We scale this value to be in the range of 0-1 with higher the values indicat-

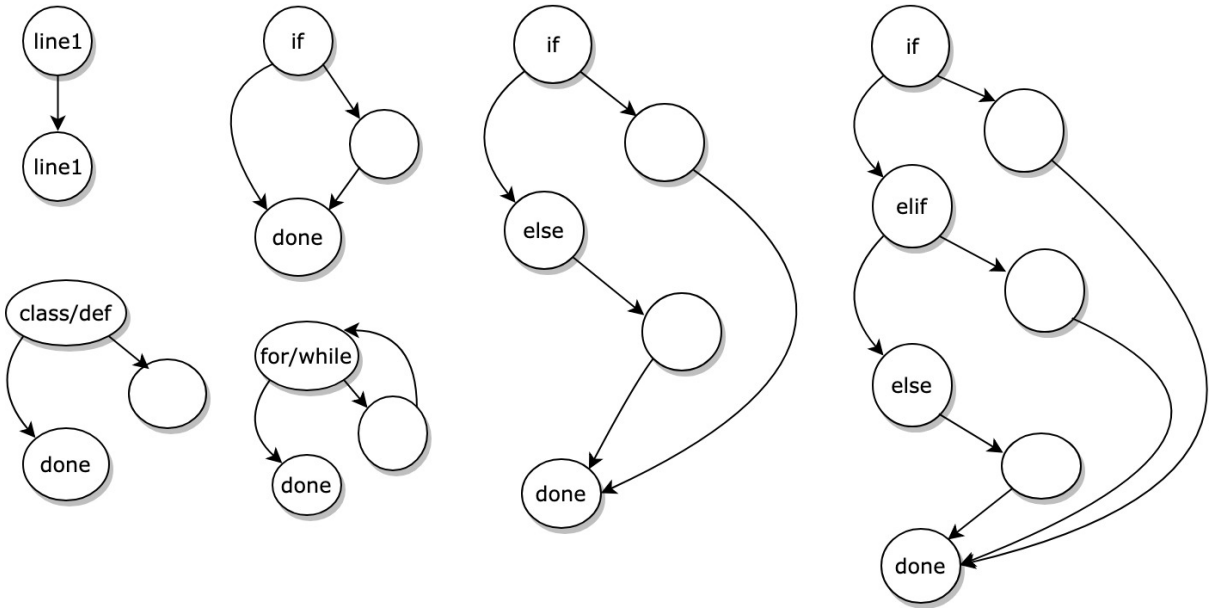

Figure 3: Control Flow diagrams for different conditions in python

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Figure 4: L2 on Eigen Energy

$$x_{ij}^{k+1} \leftarrow \frac{s_{in}^{k+1}(i,j) + s_{out}^{k+1}(i,j)}{2}.$$

Figure 5: Control Flow diagrams for different conditions in python

ing higher similarity.

- **Steady State Similarity:** To find the useful components in the graph we extracted the eigen vectors, and computed the mean squared difference on the **l2** norm of the eigen vectors (Figure 4). We scaled this value to be in the range of 0-1 with higher the values indicating higher similarity.

- **Neighbour Similarity:** Nikolic et. al. (Nikolić, 2012) proposed a technique to find the similarities of two graphs using the hubs and authorities. In this technique we compute the eigen vectors (steady state) for hub AA^T and authority A^TA and then we take the difference of their l2 norm. Figure 5 depicts the equation followed. This value is also scaled in the range of 0-1 with higher the values indicating higher similarity.

- **Graph Edit Distance:** Graph edit distance is similar to the Levenshtein distance, it is defined as the number of changes that should be made to convert graph G1 into isomorphic graph G2. This is scaled in the range of 0-1 by dividing the edit distance with maximum of sum of adjacency matrix.

5 Experimental Setup

5.1 Baselines

For the baseline models, we took the works of Ransom et. al. (Ramos et al., 2003) where they represented text in the form of vectors and applied the standard methods of measuring similarities among those vectors. Vectors are represented in the form of count vectors depicting the term frequency and also using tf-idf. Similarity measures namely - Euclidean distance similarity and cosine similarity were employed one at a time on these text representations separately.

For code matching, we employed a naive syntax matching approach of comparing two code snippets one line at a time. Two lines of code are said to be similar if the count of matching words exceeds a preset threshold.

5.2 Our Approach

Building on the existing work, models for text representation were carried forward as is. However,

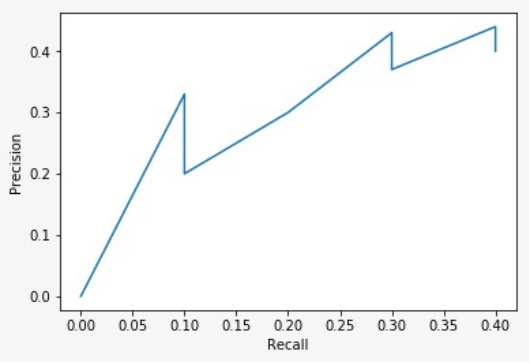

Figure 6: Precision Recall curve

evaluation of snippets of code includes matching both at a syntax and semantic level. For the syntax part, a sliding window approach of code matching was employed. At the semantic level, flow of execution of code was mimicked using control flow graphs. Overall, a weighted sum of both the approaches is taken to finalize the similarity score ranging from 0 to 1, 1 being the most similar. Figure 2 depicts the entire workflow of our model.

6 Results

For evaluation of the model, we use the Score attribute corresponding to answers. This score is the number of upvotes an answer received on StackOverflow which serves as a ground truth for our evaluation. Higher scores imply acknowledged answers by the StackOverflow community which helps us judge our model as answers with high score should appear at the top of the results. Figure 6. Following are the evaluation scores for the applied metrics on 50 samples:

Metric	Score
Precision@3	0.33
Precision@5	0.20
Recall@3	0.10
Recall@5	0.10
Precision	0.28
MAP	0.281

Table 2: Evaluation scores across metrics

7 Conclusion

In this paper, we presented a novel approach to retrieve code from stack overflow by considering many different approaches to measure the similarity of text and code. Our major contributions to the paper are as follows: (i) Explored and analysed the questions in stackoverflow for python language. (ii) Implemented a model to create a control flow and get the adjacency matrix for python codes. (iii) Introduction of multiple similarity measures that could be used together for code and text.

Reproducibility: All the codes and a sample dataset are available at https://github.com/williamscott701/stack-overflow-code-retrieval.

8 Future Work

There are couple of sections in which we can further improve our approach, few of those are as follows: (i) Consider the value of the states to measure the similarity. (ii) Explore other combinations of heuristics to find better results.

References

Miltiadis Allamanis and Charles Sutton. 2013. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 53–56. IEEE.

Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. 2012. Harnessing stack overflow for the ide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 26–30. IEEE Press.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2073–2083.

Mladen Nikolić. 2012. Measuring similarity of graph nodes by neighbor matching. *Intelligent Data Analysis*, 16(6):865–878.

Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE Press.

Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM.

Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. Piscataway, NJ.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE.

Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. 2012. Example overflow: Using social media for code recommendation. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 38–42. IEEE Press.