Scouty: A retrieval system for code pattern similarity

Luigi Ferrara | Romaniello Andrea | Bellomo Carlotta

Professors:
Toon Calders, Department of Computer Science, University of Antwerp.
Ewoenam Kwaku Tokpo, Department of Computer Science, University of Antwerp.

Department Computer Science, University of Antwerp, Middelheim 1, Antwerp, 2020, Flanders, Belgium

keywords:
Information retrieval, pattern similarity, vector space model, programming language, evaluation, python, precision, recall, inverted index, tf-idf, GitHub, Dolos.

Course: Information retrieval, 2001WETINR
Academic year: 2023 - 2024

1 | INTRODUCTION

Imagine being a programmer, engrossed in the development of a software project. After a while, you build up some code that works as expected.
Now, you would like to check if the snippet you have developed is well implemented, if there are better ways to implement it, if it can be improved, if it can run faster, or if it can be written with fewer lines of code. You can definitely exploit ChatGPT or other similar tools to assist you in this task. But let's say you are seeking something better.

As a fully-skilled software developer, you are completely aware that ChatGPT may fail to determine the best answer to a code query.
A possible solution is checking on GitHub for stable and well-structured repositories to find some code similar to yours and see if it has been developed better.

The process I explained right now seems to be very time-consuming, and if you don't know exactly where to look, finding relevant information may prove challenging.

That's where our system comes into play.

1.1 | RESEARCH GOALS

Navigating through GitHub for optimal code can indeed be intricate, especially without a clear direction. Our solution aims to make this process simpler and automatic, offering a more efficient and effective way for developers to discover well-implemented code snippets thanks to our system.

The primary research goal of this project is to create a tool that speeds up the process of finding code snippets on GitHub using a **vector space model** for information retrieval. This involves the key task of vectorising both the user's code query and the code snippets in our GitHub dataset.
By converting these elements into vector forms, we facilitate a more effective comparison and matching process. Our focus is on how **accurately** and **efficiently** the tool can compare the query vector with the vectors of the code snippets, thereby identifying the most relevant matches. The aim is to understand the effectiveness of the vector space model in interpreting and retrieving code, considering factors like accuracy, speed, and relevance of the results. This research will contribute to a deeper understanding of information retrieval in software development and could significantly improve how developers find and utilise existing code, ultimately enhancing productivity.

This text examines the following research questions:

- **RQ1**: can the vector space model be used in order to build a system that aims to solve this problem ?

- **RQ2**: can the system be tested to understand if it works properly?

RQ1 and RQ2 are addressed in the next sections, where we provide detailed responses and insights.

1.2 | CLARIFICATIONS

In this paper, we are not suggesting avoiding the use of ChatGPT at all, since the vast majority of programmers use it in various contexts, such as to solve program ming bugs (S. Surameery, Y. Shakor, 2023[1]) or to accelerate their learning process as students (M. Rahman, Y. Watanobe, 2023[2]). Nevertheless, our focus shifts to a different scenario.

It is worth noting that some students also admit to using it for cheating (C. Westfall, 2023[3]).

2 | RELATED WORKS

In the context of code pattern similarities, our system operates in a manner analogous to a plagiarism detection system. A similar research for code plagiarism has been done by Deniz Kılınç, Fatma Bozyiğit, Alp Kut, Muhammet Kaya[4]. This study is particularly valuable for our project because it explores the use of vector space models to detect plagiarism in source code.

This approach is very similar to the methodology used by Scouty, which also employs vector space models for retrieving code patterns. Another study on using VSM for Bug Localisation was done by Singapore Management University (Shaowei Wang, David Lo, and

Julia Lawall[5]). This research underlines the potential of VSMs as powerful tools for pattern retrieval in complex software systems, reinforcing the viability of VSMs for similar applications in our project.

The work done by Sushil Bajracharya, Joel Ossher, C. Lopes[6] represents an alternative approach to the goal of our project. While this study uses Structural Semantic Indexing (SSI) to enhance the retrieval of API usage examples. Including this study in our related works section acknowledges a different perspective on achieving similar goals in code retrieval, underscoring the diversity of methods available and our deliberate choice to focus on VSMs for their particular strengths and capabilities.

2.1 | DOLOS

For the evaluation, we needed to find a stable system to compare the results. Dolos, Developed at Ghent University, is part of an active research initiative.

Dolos[7] is accessible as both an online service and a command-line tool, offering flexibility for different user preferences. This multifaceted approach to plagiarism detection, especially its ability to handle multiple languages, makes Dolos a significant reference in the realm of code analysis and relevant to our project's evaluation.

Dolos is an open-source tool developed at the University of Ghent which detect the **plagiarised code**.
The inspiration for its creation stems from the recognition that is common for students to copy code at least once during their academic journey.

Developed using TypeScript, Dolos is made available to the community under the permissive MIT open-source license. The code is detected by using the tree-sitter parsing library with state-of-the-art string-matching algorithms to detect similar code fragments (R. Maertens, C. V. Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt, Bart Mesuere[8]).

Dolos can work with both plain text files and structured programming language files.

Dolos' algorithm consists of four key steps:

Tokenisation: using the tree-sitter parsing library, Dolos transforms each source file into an abstract syntax tree (AST). In this tree, each node signifies a distinct code construct. Subsequently, each AST undergoes serialisation into a list of tokens. Each AST is serialised in a list of tokens. In this way, the structure of the code is preserved but formatting, variable names, delimiters, comments, etc. are deleted. In this way, similarity detection results are more robust against many types of obfuscations that are commonly found in educational source code plagiarism.

Fingerprinting: the k-grams (common sequences of k successive tokens) are searched. To speed up the searching phase, tokens and k-grams are represented as integers.

Indexing: in the previous step each source file has been converted into a filtered list of fingerprints. to speed up the process of finding common fingerprints, an index that maps each fingerprint onto all its occurrences in the source file has been developed. For each index, the corresponding k-gram location is stored in the source file. In this way, the index contains all the information needed to compare the files and visualise shared fragments.

Reporting: three plagiarism metrics are developed in Dolos. The metrics represent different views on which shared fingerprints are relevant and how to quantify them.

While developing Scouty, we considered the mechanisms already integrated into Dolos and we adapted some of them.

3 | SCOUTY

Our system has already been introduced, so we will promptly go into the **implementation** of the fundamental portions that better describe what Scouty does.

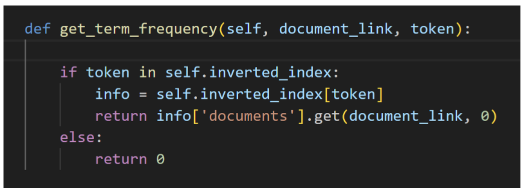
3.1 | INDEXING

Our indexing implementation is designed to enable fast and efficient document retrieval, making it crucial for managing and accessing large datasets.
An **inverted index** is a database index that maps content, such as words or numbers, to its location in a database file or documents. Essentially, it is a hashmap-like data structure that directs you from a word, known as token or term, to a document.

In our implementation, we used a '*Dataset*' class to systematically update the index during a for loop, where each iteration processes a document and its associated tokens.

The index is stored as a **Python dictionary**, where each key represents a term and its value is a nested dictionary containing documents with that term. This nested structure includes a term frequency count for each document, allowing for constant-time retrieval of document information, regardless of the dataset size.

Example of constant operation to retrieve the term frequency for a specific token in a given document



```python
def get_term_frequency(self, document_link, token):

    if token in self.inverted_index:
        info = self.inverted_index[token]
        return info['documents'].get(document_link, 0)
    else:
        return 0
```

Additionally, the index includes functions for retrieving documents containing a specific token, document frequency of a token, term frequencies in a document, and the total number of documents indexed, providing comprehensive search capabilities.

3.2 | SOURCE CODE VECTORISATION

Vectorising source code is a crucial step in various applications, such as code similarity analysis, plagiarism detection, and even automated code generation and refactoring. The process of transforming raw source code into a mathematical representation that can be easily processed and compared using algorithms is known as vectorisation.

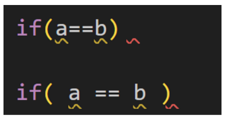
We took a particular approach to vectorisation for source code:

Removal of Comments: we have a *parser.remove_comments()* function that strips the code of comments. Comments usually contain natural language descriptions and don't contribute to the logical or structural essence of the code.

By removing them, we focus the analysis on the functional part of the code, which is more relevant for our purpose.

Adjusting Spaces Around Operators and Punctuations: In the first version of our program, an oversight was identified in the tokenisation process, particularly in the treatment of whitespace characters.

We used a basic method called *string.split()*, which separates the code based on spaces. However, this method proved to be too simple because it only looks for spaces to split the code. This led to a significant issue: two pieces of code that were essentially the same were treated as different just because they had spaces in different places.



```
if(a==b)

if( a == b )
```

Both lines perform the same operation, but our original tokenising method viewed them as different due to the varying use of spaces. This inconsistency became a problem, especially when calculating the importance of specific code segments TF-IDF.

To address this, we developed a specialised function that standardises spaces around certain parts of the code.

The *parser.adjust_spaces()* function adds spaces around operators and punctuation. This standardisation is crucial for tokenisation, as it ensures that operators and punctuation marks are correctly identified as separate tokens. Consistent tokenisation is essential for the VSM, as it relies on the presence or absence of these tokens to understand and compare code snippets.

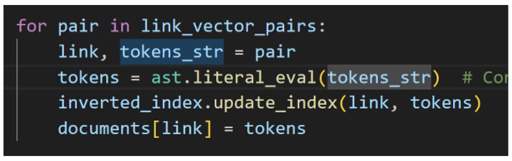
Tokenisation and Filtering: The final step, carried out in the vectorise function, is to split the code into tokens and filter out specific keywords and punctuation. This step is significant because:

- It reduces the complexity of the code representation by removing common language constructs (like control flow keywords if, else, for, etc.), which might not contribute significantly to the uniqueness or specific functionality of a code snippet.

- By focusing on more distinctive elements (like variable names, function names, and unique operators), the vectorised form of the code is more likely to capture its unique characteristics.

The final step of our process produces a list of carefully selected tokens, ready to be converted into vectors. Once we finish processing all the documents, we save them in a JSON file. This step is important because it means we don't have to process the same documents every time the program starts, saving time and resources.

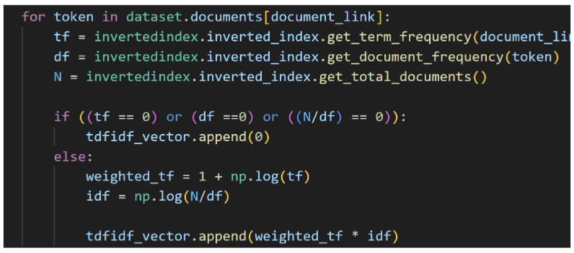
When the program starts, each token from the documents is added to an inverted index. Alongside this, we create a global dictionary that links each document's unique URL or ID with its tokens. This setup is efficient for quickly accessing the data and is useful for creating the final vector.



```
for pair in link_vector_pairs:
    link, tokens_str = pair
    tokens = ast.literal_eval(tokens_str)  # Co
    inverted_index.update_index(link, tokens)
    documents[link] = tokens
```

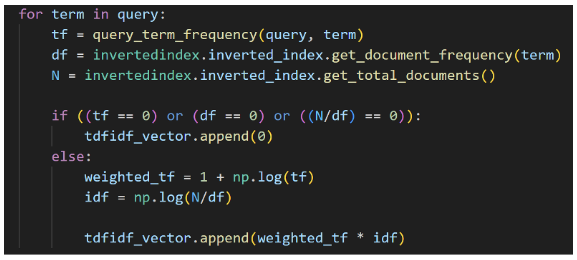
3.3 | TF-IDF VECTOR NORMALISATION

The *ranking.transform_to_non_normalized_tfidf()* function is designed to compute the Term Frequency-Inverse Document Frequency (TF-IDF) vector for a given document in its non-normalised form. The function calculates the TF (Term Frequency) and IDF (Inverse Document Frequency) for each term in the document. TF measures how frequently a term appears in the document, while IDF gives more weight to terms that are rare across the document set.



```
for token in dataset.documents[document_link]:
    tf = invertedindex.inverted_index.get_term_frequency(document_li
    df = invertedindex.inverted_index.get_document_frequency(token)
    N = invertedindex.inverted_index.get_total_documents()

    if ((tf == 0) or (df ==0) or ((N/df) == 0)):
        tdfidf_vector.append(0)
    else:
        weighted_tf = 1 + np.log(tf)
        idf = np.log(N/df)

        tdfidf_vector.append(weighted_tf * idf)
```

“Function to convert documents tokens to normalised vectors”



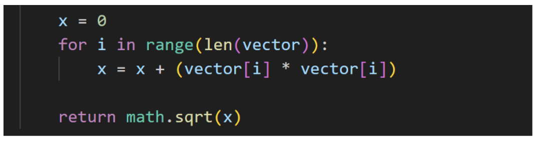
```python
for term in query:
    tf = query_term_frequency(query, term)
    df = invertedindex.inverted_index.get_document_frequency(term)
    N = invertedindex.inverted_index.get_total_documents()

    if ((tf == 0) or (df == 0) or ((N/df) == 0)):
        tdfidf_vector.append(0)
    else:
        weighted_tf = 1 + np.log(tf)
        idf = np.log(N/df)

        tdfidf_vector.append(weighted_tf * idf)
```

“Function to convert query tokens to normalised vectors”

Each document vector will be different in length. We need to normalise them to do operations. Normalisation, in this context, refers to adjusting the values of the vector so that they collectively have a unit length.

It involves dividing each term's TF-IDF score by the Euclidean norm of the entire vector. This adjusts the vector to a standard scale without distorting differences in the values of the scores.

For instance, when computing the similarity between documents using cosine similarity, normalised vectors are essential. This is because cosine similarity focuses on the angle between vectors, not their magnitude. Normalised vectors ensure that this angle accurately reflects the similarity in terms of content, irrespective of the length of the documents.



```python
    x = 0
    for i in range(len(vector)):
        x = x + (vector[i] * vector[i])

    return math.sqrt(x)
```

“Function to normalise the vector”

3.4 | SCORING THE DOCUMENTS

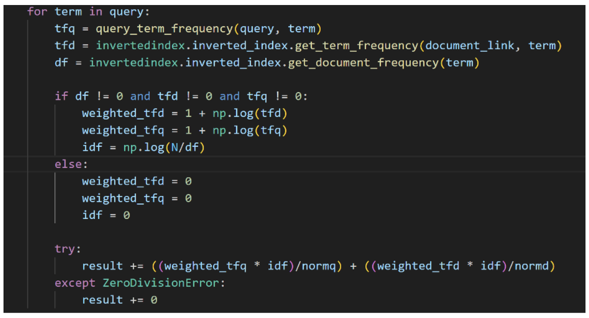
In our scoring process, first, we get the term frequency (TF) for both the document and the query using an inverted index, which is efficiently facilitated by constant-time operations due to the structure implementation.

We apply a logarithmic scale to these frequencies, which helps in balancing the importance of the terms.

Next, we calculate the Inverse Document Frequency (IDF). This is done by taking the logarithm of the total number of documents (N) divided by the number of documents that contain the term (df).

This step helps us understand how rare or common a term is across all documents.

Finally, with this information, we calculate a score for each document. This score combines the term frequencies and their rarity (IDF) to measure how relevant each document is to the query.



```python
for term in query:
    tfq = query_term_frequency(query, term)
    tfd = invertedindex.inverted_index.get_term_frequency(document_link, term)
    df = invertedindex.inverted_index.get_document_frequency(term)

    if df != 0 and tfd != 0 and tfq != 0:
        weighted_tfd = 1 + np.log(tfd)
        weighted_tfq = 1 + np.log(tfq)
        idf = np.log(N/df)
    else:
        weighted_tfd = 0
        weighted_tfq = 0
        idf = 0

    try:
        result += ((weighted_tfq * idf)/normq) + ((weighted_tfd * idf)/normd)
    except ZeroDivisionError:
        result += 0
```

4 | INFORMATION RETRIEVAL TOPICS

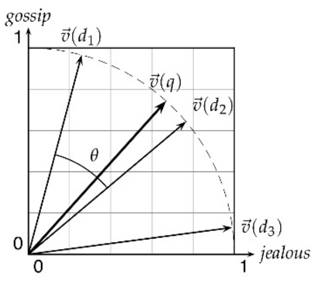
To gain a better understanding of our system, we will thoroughly examine various information retrieval topics.

4.1 | VECTOR SPACE MODEL

In this context, the vector space model employs a mathematical technique to represent documents and queries as vectors. Each dimension within this multi-dimensional space corresponds to a unique term.

In the vector space model:

- Each document is represented as a vector in a high-dimensional space. The dimension of the vector corresponds to terms in the entire collection of documents.
- Each unique term in the document collection represents a dimension in the vector space. the value of each dimension is typically a weight that represents the importance of that term in the document.
- A vector of weighted terms represents a document. is possible to compute the weights using various methods.



We opted to use a vector space model to determine the similarity between documents and query.

The standard way of quantifying the similarity between 2 documents is to compute the cosine similarity of their vector representations, i.e.,

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|},$$

Where the numerator represents the *dot product* (also known as the inner product) of the vectors $\vec{V}$(d1) and $\vec{V}$(d2), while the denominator is the product of their *Euclidean lengths*.

4.2 | TF-IDF SCORING

TF-IDF (Term Frequency - Inverse Document Frequency) is a measure of importance of a word to a document in a collection.
Specifically, our approach involves utilising a weighted version of this metric known as weighted TF-IDF, i.e,

$$\text{wf-idf}_{t,d} = \text{wf}_{t,d} \times \text{idf}_t.$$

Where:

$$\text{wf}_{t,d} = \begin{cases} 1 + \log \text{tf}_{t,d} & \text{if } \text{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}.$$

$$\text{idf}_t = \log \frac{N}{\text{df}_t}.$$

In other words, this formula assigns a weight t to a document d that is:

- High when t occurs many times within a small number of documents (thus landing high discriminating power to those documents)

- Lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronouncing value)

- The lowest when the term occurs in virtually all the documents

In the end the similarity score between a query and a certain document is:

$$\text{Score}(q, d) = \sum_{t \in q} \text{wtf-idf}_{t,d}.$$

Transitioning to a more detailed formulation, let's see a more explicit version of this formula:

$$\sum_{t \in q} \frac{\log\left(\frac{N}{df_t}\right)(1+\log(tf_{t,d}))}{norm_d} \times \frac{\log\left(\frac{N}{df_t}\right)(1+\log(tf_{t,q}))}{norm_q}$$

4.3 | INVERTED INDEX

We had to implement a dynamic inverted index in our system. This decision stems from our requirement to update the index periodically.

Given that our dataset comprises numerous repository projects, the introduction of new documents may occur over time within these projects. To accommodate these changes, we have integrated an update method into the system.

This update method involves the manual retrieval of new documents, reflecting the infrequent nature of updates, as the repositories we analyse typically belong to stable projects. However, despite project stability, occasional changes may transpire.
Consequently, our adapted version of the traditional inverted index is designed to handle such possibilities. Our implementation of the inverted index prioritises flexibility and dynamism.

The system enables us to retrieve documents at our discretion, aligning with our commitment to adaptability based on varying needs. This feature highlights the versatility of our traditional inverted index, which does not mandate constant retrieval and can be customised to meet specific project requirements.

4.4 | EVALUATION

Usually, to evaluate an information retrieval system, metrics like *Precision* and *Recall* are used. To calculate them is necessary to know which documents are relevant and which are not.

To determine if a document is relevant or not, it is crucial to know if it addresses the stated information need.

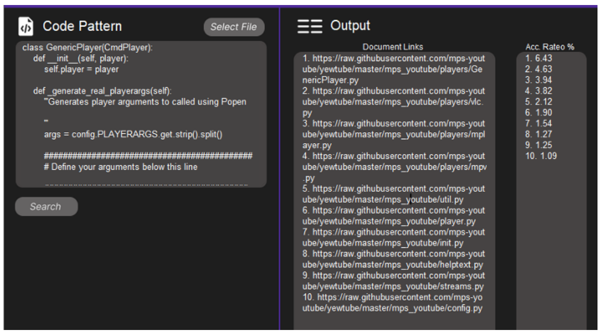
An *information need* is not simply a query, but more like the ultimate intention of a user. A document addresses a stated information need not merely due to the presence of all the query words.

The stated information is often not explicitly expressed and may not be easily discernible.

5 | EVALUATION OF THE SYSTEM

As previously mentioned, we opted to use Dolos as a comparative system, allowing us to evaluate Scouty by comparing the results from both systems.

5.1 | FIRST TEST

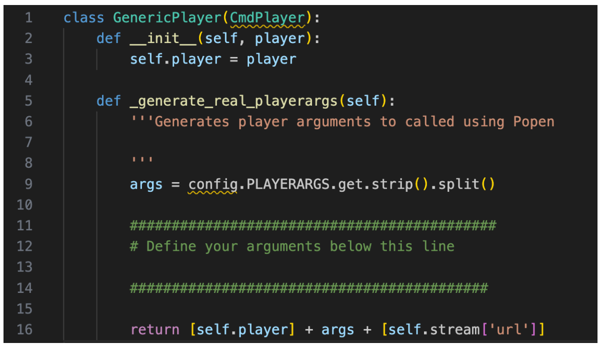


```
class GenericPlayer(CmdPlayer):
    def __init__(self, player):
        self.player = player

    def _generate_real_playerargs(self):
        '''Generates player arguments to called using Popen

        -
        args = config.PLAYERARGS.get.strip().split()

        ############################################
        # Define your arguments below this line
```

```
Document Links                                        Acc. Rateo %
1. https://raw.githubusercontent.com/mps-yout         1. 6.43
ube/yewtube/master/mps_youtube/players/Ge             2. 4.63
nericPlayer.py                                        3. 3.94
2. https://raw.githubusercontent.com/mps-yout         4. 3.82
ube/yewtube/master/mps_youtube/players/vlc.           5. 2.12
py                                                    6. 1.90
3. https://raw.githubusercontent.com/mps-yout         7. 1.54
ube/yewtube/master/mps_youtube/players/mpl            8. 1.27
ayer.py                                               9. 1.25
4. https://raw.githubusercontent.com/mps-yout         10. 1.09
ube/yewtube/master/mps_youtube/players/mpv
.py
5. https://raw.githubusercontent.com/mps-yout
ube/yewtube/master/mps_youtube/util.py
6. https://raw.githubusercontent.com/mps-yout
ube/yewtube/master/mps_youtube/player.py
7. https://raw.githubusercontent.com/mps-yout
ube/yewtube/master/mps_youtube/init.py
8. https://raw.githubusercontent.com/mps-yout
ube/yewtube/master/mps_youtube/helptext.py
9. https://raw.githubusercontent.com/mps-yout
ube/yewtube/master/mps_youtube/streams.py
10. https://raw.githubusercontent.com/mps-yo
utube/yewtube/master/mps_youtube/config.py
```

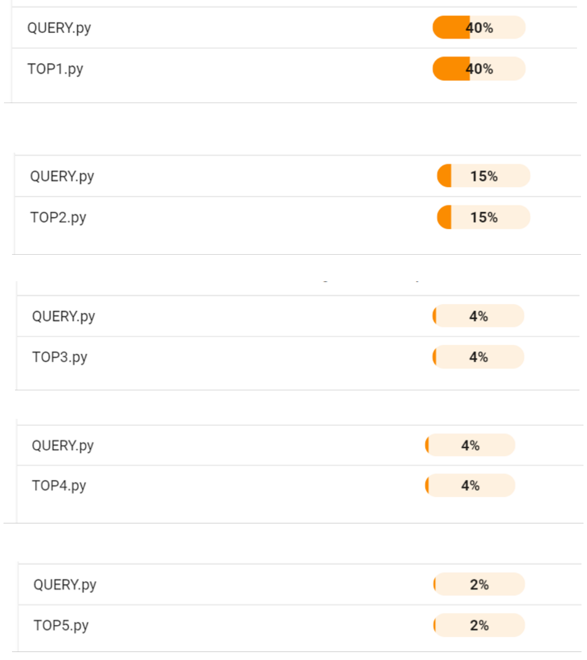
"A screenshot of our Scouty at work"

From our results, file 1 and 2 should have a good similarity. We checked the top 5 files with Dolos.



```python
class GenericPlayer(CmdPlayer):
    def __init__(self, player):
        self.player = player

    def _generate_real_playerargs(self):
        '''Generates player arguments to called using Popen

        '''
        args = config.PLAYERARGS.get.strip().split()

        ############################################
        # Define your arguments below this line

        ############################################

        return [self.player] + args + [self.stream['url']]
```

"Tested query"



QUERY.py	40%
TOP1.py	40%

QUERY.py	15%
TOP2.py	15%

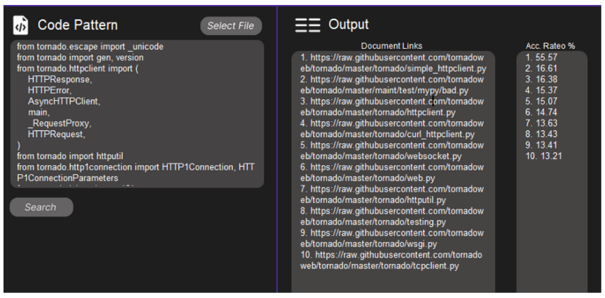
QUERY.py	4%
TOP3.py	4%

QUERY.py	4%
TOP4.py	4%

QUERY.py	2%
TOP5.py	2%

5.2 | SECOND TEST

The second approach was about testing a query that was exactly equal to a document in the dataset, i.e, https://raw.githubusercontent.com/tornadoweb/tornado/master/tornado/simple_httpclient.py. In this scenario, the system should return an high score exclusively for the first entry.

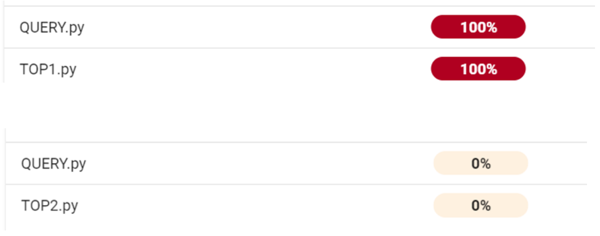


```
from tornado.escape import _unicode
from tornado import gen, version
from tornado.httpclient import (
    HTTPResponse,
    HTTPError,
    AsyncHTTPClient,
    main,
    _RequestProxy,
    HTTPRequest,
)
from tornado import httputil
from tornado.http1connection import HTTP1Connection, HTT
P1ConnectionParameters
```

```
Document Links                                          Acc. Rateo %
1. https://raw.githubusercontent.com/tornadow           1. 55.57
eb/tornado/master/tornado/simple_httpclient.py          2. 16.61
2. https://raw.githubusercontent.com/tornadow           3. 16.38
eb/tornado/master/maint/test/mypy/bad.py                4. 15.37
3. https://raw.githubusercontent.com/tornadow           5. 15.07
eb/tornado/master/tornado/httpclient.py                 6. 14.74
4. https://raw.githubusercontent.com/tornadow           7. 13.63
eb/tornado/master/tornado/curl_httpclient.py            8. 13.43
5. https://raw.githubusercontent.com/tornadow           9. 13.41
eb/tornado/master/tornado/websocket.py                  10. 13.21
6. https://raw.githubusercontent.com/tornadow
eb/tornado/master/tornado/web.py
7. https://raw.githubusercontent.com/tornadow
eb/tornado/master/tornado/httputil.py
8. https://raw.githubusercontent.com/tornadow
eb/tornado/master/tornado/testing.py
9. https://raw.githubusercontent.com/tornadow
eb/tornado/master/tornado/wsgi.py
10. https://raw.githubusercontent.com/tornado
web/tornado/master/tornado/tcpclient.py
```

"A screenshot of our Scouty at work"

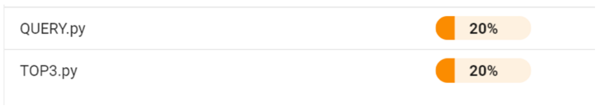
As we can see, the first document exceeds the average score by a lot. Let's compare the top 5 result in Dolos. As we expected, the first document is 100% same since we used it for the query



QUERY.py	100%
TOP1.py	100%

QUERY.py	0%
TOP2.py	0%

The second result is a complete different file. Our system scores it because it contains some common words.

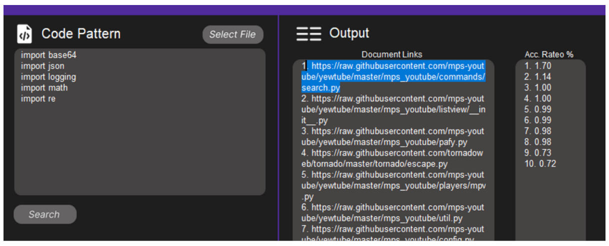


QUERY.py	20%
TOP3.py	20%

20% is not significantly big compared to 100% from the first one. We want now to use a more generic query to retrieve all the files that imports same libraries.

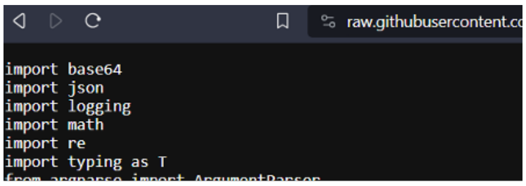
5.2 | THIRD TEST

This time we aimed to test a query with a high probability to be in several documents.



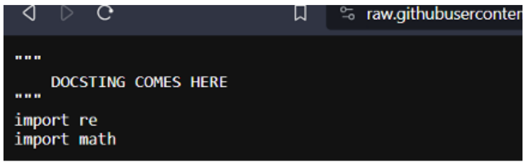
```
import base64
import json
import logging
import math
import re
```

```
Document Links                                          Acc. Rateo %
1. https://raw.githubusercontent.com/mps-yout           1. 1.70
ube/yewtube/master/mps_youtube/commands/                2. 1.14
search.py                                               3. 1.00
2. https://raw.githubusercontent.com/mps-yout           4. 1.00
ube/yewtube/master/mps_youtube/listview/__in            5. 0.99
it__.py                                                 6. 0.99
3. https://raw.githubusercontent.com/mps-yout           7. 0.98
ube/yewtube/master/mps_youtube/pafy.py                  8. 0.98
4. https://raw.githubusercontent.com/tornadow           9. 0.73
eb/tornado/master/tornado/escape.py                     10. 0.72
5. https://raw.githubusercontent.com/mps-yout
ube/yewtube/master/mps_youtube/players/mpv
.py
6. https://raw.githubusercontent.com/mps-yout
ube/yewtube/master/mps_youtube/util.py
7. https://raw.githubusercontent.com/mps-yout
```



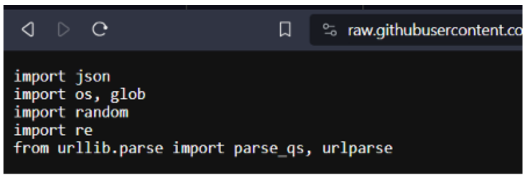
```
import base64
import json
import logging
import math
import re
import typing as T
from argparse import ArgumentParser
```

“First result”



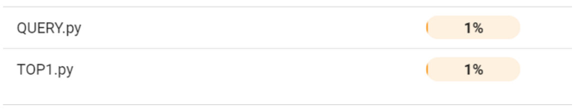
```
"""
    DOCSTING COMES HERE
"""
import re
import math
```

“Second result”

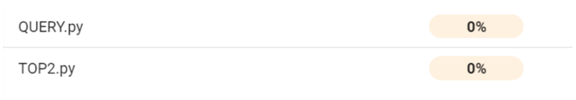


```
import json
import os, glob
import random
import re
from urllib.parse import parse_qs, urlparse
```

“Third result”



QUERY.py	1%
TOP1.py	1%



QUERY.py	0%
TOP2.py	0%

Dolos score is very low since we are not using code at all. There is no flow in the code we gave, its just based on similarity of imports.

6 | CONCLUSION

Our system is really good at finding code that is similar or the same, or at identifying code that uses certain APIs. It focuses on the way the code is written - its syntax - but doesn't look at what the code actually does, or its flow. To understand the flow or the deeper meaning of the code, we would need to use more complex methods like Abstract Syntax Trees (AST) or artificial intelligence.

This approach is especially useful for spotting code that has been copied and pasted with only small changes. It's also good at finding specific examples of how APIs are used. However, it's important to remember that our system is limited to analysing the structure and format of the code, not what the code means or how it works. For a deeper understanding of the code's functionality, more advanced techniques would be necessary.

The text above answer to RQ1.

With regard to the RQ2, we have done several tests using Dolos as a reference system.
This approach ensures a thorough examination, even though Dolos uses different metrics and methodologies for evaluating comparable code snippets.

“Any non-trivial semantic property of a language which is recognised by a Turing machine is undecidable”
Henry Gordon Rice, 1953

[1] Nigar M. Shafiq Surameery, Mohammed Y. Shakor (2023). Use ChatGPT To Solve Programming Bugs. https://journal.hmjournals.-com/index.php/IJITC

[2] Md. Mostafizer Rahman and Yukuta Watanobe (2023). ChatGPT for Education and Research: Opportunities, Threats, and Strategies. https://www.mdpi.com/2076-3417/13/9/5783

[3] Chris Westfall (2023). Forbs. Educators Battle Plagiarism As 89% Of Students Admit To Using OpenAI's ChatGPT For Homework. https://www.forbes.com/sites/chriswestfall/2023/01/28/educators-battle-plagiarism-as-89-of-stu-dents-admit-to-using-open-ais-chatgpt-for-homework/?sh=5bb37e98750d

[4] Deniz Kılınç, Fatma Bozyiğit, Alp Kut, Muhammet Kaya (2015). Overview of Source Code Plagiarism In Program-ming Courses. https://www.researchgate.net/publication/275713992_Overview_of_Source_Code_Plagiarism_in_Pro-gramming_Courses

[5] Shaowei Wang, David Lo, and Julia Lawall (2014). Compositional Vector Space Model for Bug Localisation. https://ieeexplore.ieee.org/abstract/document/6976083?casa_token=Gmg08rTzkFUAAAAA:ffWCZkS1quKAMtDCFP3DFe-XvLMeUNUaLQzfMwE_aj41G1E-04Uk23aDTkTM6apdEhLxXkjSM1G6w

[6] Sushil Bajracharya, Joel Ossher, C. Lopes (2010). Leveraging usage similarity for effective retrieval of examples in code repositories. https://dl.acm.org/doi/pdf/10.1145/1882291.1882316

[7] DOLOS. https://dolos.ugent.be/

[8] Dolos: Language-agnostic plagiarism detection in source code (2022). https://onlinelibrary.wiley.com/doi/abs/10.1111/jcal.12662