Information Retrieval - Search Engine

Ράτσικας Αθανάσιος Μάριος – inf2021193
Αβαγιανός Μιχαήλ – inf2021009

Abstract

This project has been developed and tested in the following environments:

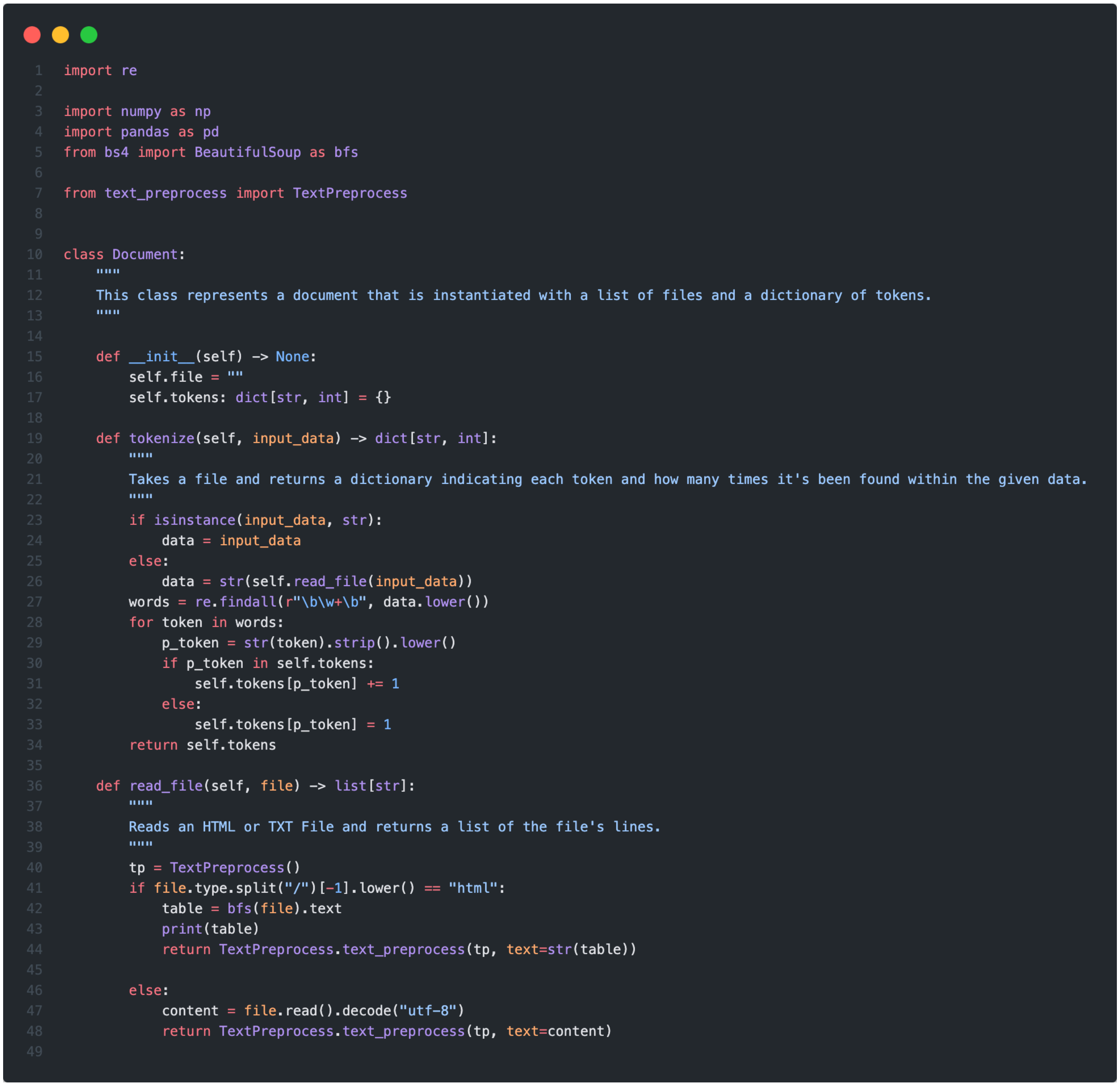
- **MacOS 14.5.0 (23F79)** – aarch64
- **ZorinOS 17.1 Core** – x86_64

We use Python v3.11.2 for this project.

Every change has been recorded on a private Github repository, which we can always give you access to if needed.

To run every file and have the results printed in the console & a results.txt file, you will need to run **streamlit run ./src/Main.py**.

Document Class

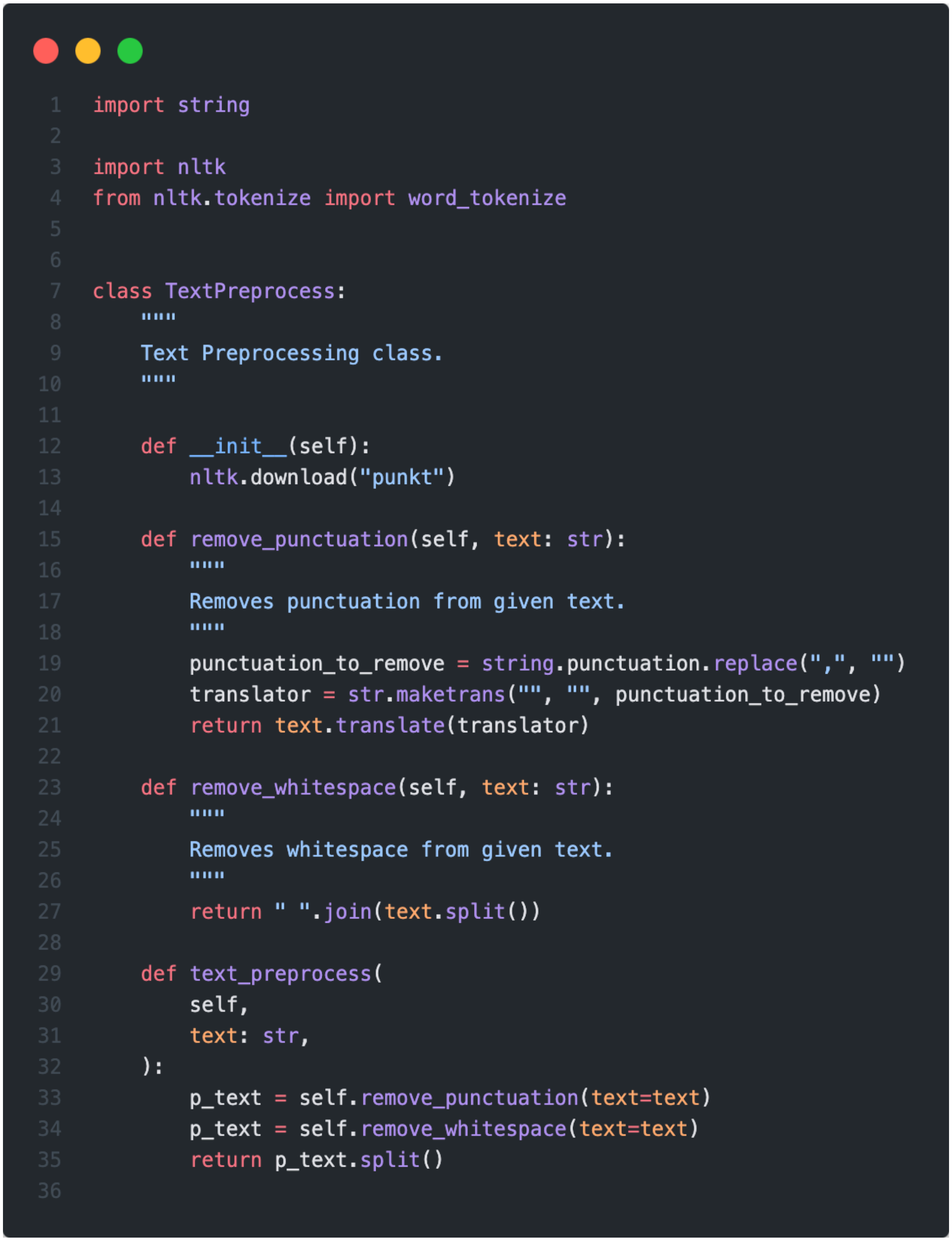


```
1   import re
2
3   import numpy as np
4   import pandas as pd
5   from bs4 import BeautifulSoup as bfs
6
7   from text_preprocess import TextPreprocess
8
9
10  class Document:
11      """
12      This class represents a document that is instantiated with a list of files and a dictionary of tokens.
13      """
14
15      def __init__(self) -> None:
16          self.file = ""
17          self.tokens: dict[str, int] = {}
18
19      def tokenize(self, input_data) -> dict[str, int]:
20          """
21          Takes a file and returns a dictionary indicating each token and how many times it's been found within the given data.
22          """
23          if isinstance(input_data, str):
24              data = input_data
25          else:
26              data = str(self.read_file(input_data))
27          words = re.findall(r"\b\w+\b", data.lower())
28          for token in words:
29              p_token = str(token).strip().lower()
30              if p_token in self.tokens:
31                  self.tokens[p_token] += 1
32              else:
33                  self.tokens[p_token] = 1
34          return self.tokens
35
36      def read_file(self, file) -> list[str]:
37          """
38          Reads an HTML or TXT File and returns a list of the file's lines.
39          """
40          tp = TextPreprocess()
41          if file.type.split("/")[-1].lower() == "html":
42              table = bfs(file).text
43              print(table)
44              return TextPreprocess.text_preprocess(tp, text=str(table))
45
46          else:
47              content = file.read().decode("utf-8")
48              return TextPreprocess.text_preprocess(tp, text=content)
49
```

Figure I *- The* ***document.py*** *file.*

As seen in **Figure I**, this file represents the **Document** class, which we use to instantiate all the input files (html or text), and implement tokenization and file reading. We have had some issues reading html files, but have found out how to properly handle them using regex and later found that bfs has a nice **text** attribute (found on line 42), that automatically takes all of the text from an html file.

TextPreprocess Class

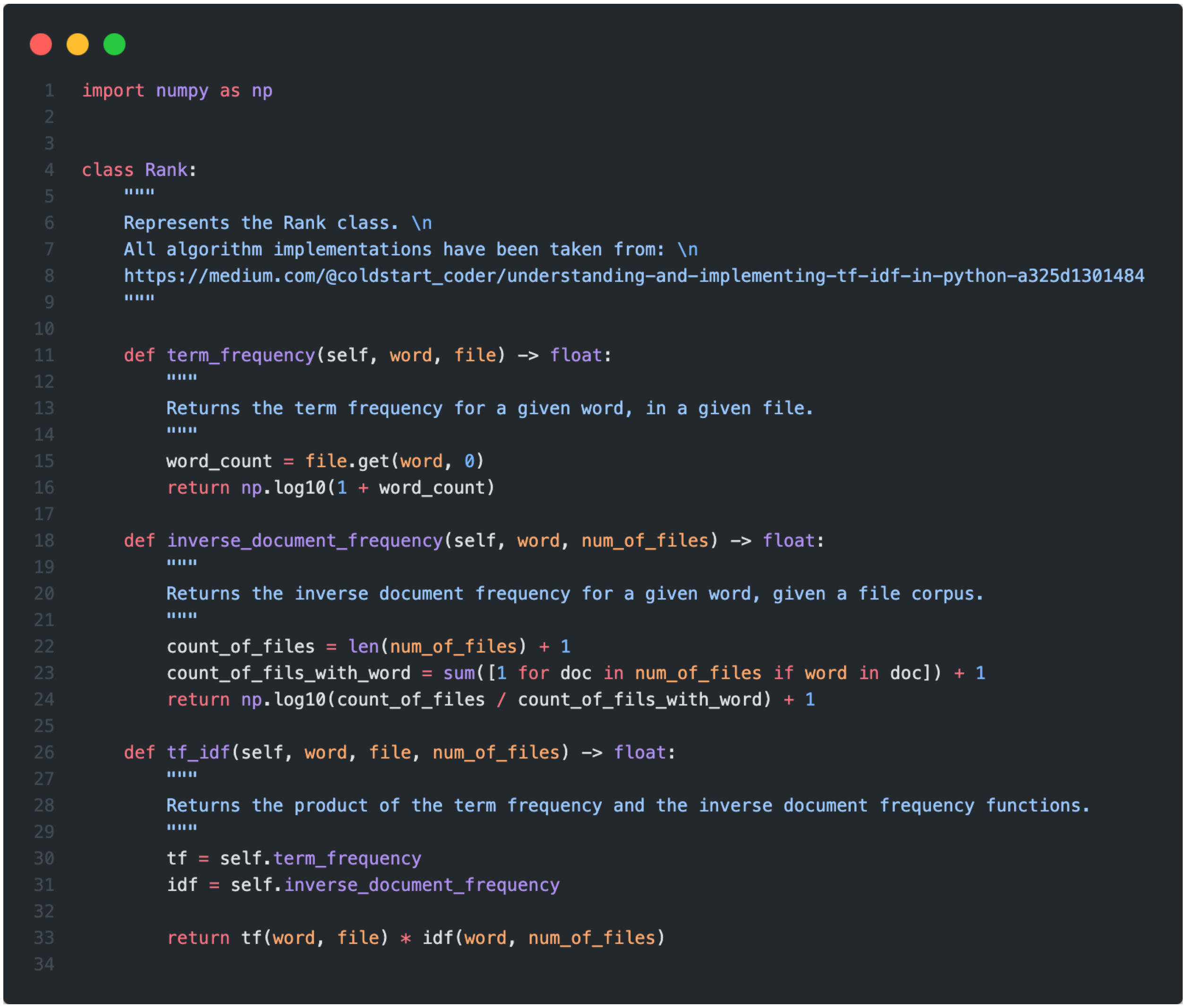


```python
import string

import nltk
from nltk.tokenize import word_tokenize


class TextPreprocess:
    """
    Text Preprocessing class.
    """

    def __init__(self):
        nltk.download("punkt")

    def remove_punctuation(self, text: str):
        """
        Removes punctuation from given text.
        """
        punctuation_to_remove = string.punctuation.replace(",", "")
        translator = str.maketrans("", "", punctuation_to_remove)
        return text.translate(translator)

    def remove_whitespace(self, text: str):
        """
        Removes whitespace from given text.
        """
        return " ".join(text.split())

    def text_preprocess(
        self,
        text: str,
    ):
        p_text = self.remove_punctuation(text=text)
        p_text = self.remove_whitespace(text=text)
        return p_text.split()

```

***Figure II -** The **text_preprocess.py** file.*

Likewise, in **Figure II**, we define the **TextPreprocess** class, which helps us remove punctuation and whitespace from the text, during a document's tokenization.

Rank Class

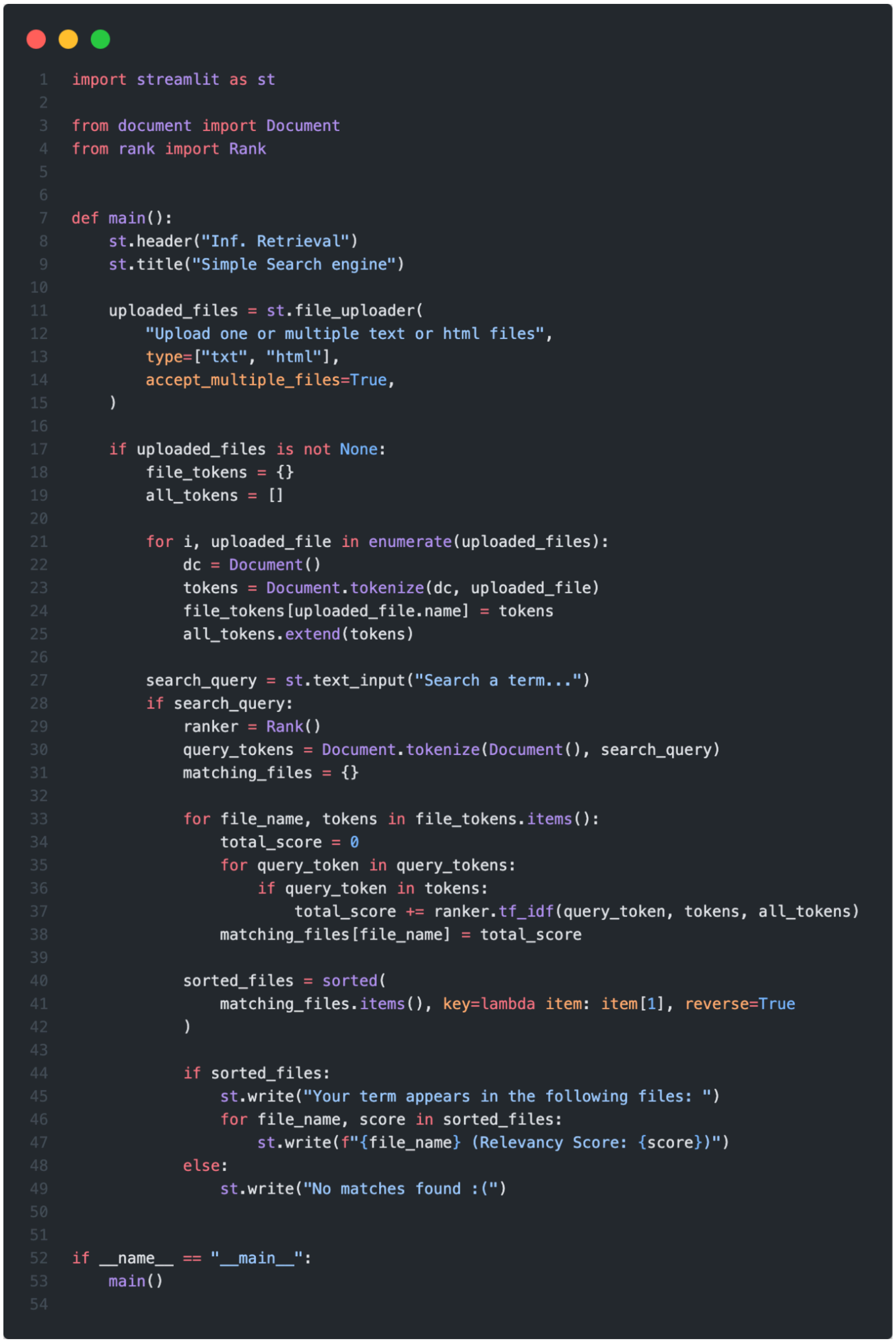


```python
import numpy as np


class Rank:
    """
    Represents the Rank class. \n
    All algorithm implementations have been taken from: \n
    https://medium.com/@coldstart_coder/understanding-and-implementing-tf-idf-in-python-a325d1301484
    """

    def term_frequency(self, word, file) -> float:
        """
        Returns the term frequency for a given word, in a given file.
        """
        word_count = file.get(word, 0)
        return np.log10(1 + word_count)

    def inverse_document_frequency(self, word, num_of_files) -> float:
        """
        Returns the inverse document frequency for a given word, given a file corpus.
        """
        count_of_files = len(num_of_files) + 1
        count_of_fils_with_word = sum([1 for doc in num_of_files if word in doc]) + 1
        return np.log10(count_of_files / count_of_fils_with_word) + 1

    def tf_idf(self, word, file, num_of_files) -> float:
        """
        Returns the product of the term frequency and the inverse document frequency functions.
        """
        tf = self.term_frequency
        idf = self.inverse_document_frequency

        return tf(word, file) * idf(word, num_of_files)

```

***Figure III -** The **rank.py** file.*

Finally, in **Figure III**, we define the **Rank** class, which handles all of the ranking algorithms. We have implemented TF, IDF, and TF-IDF - which we use in the ranking process - and have all been documented, like all previous files, as shown in the figures.

Main Class



```python
import streamlit as st

from document import Document
from rank import Rank


def main():
    st.header("Inf. Retrieval")
    st.title("Simple Search engine")

    uploaded_files = st.file_uploader(
        "Upload one or multiple text or html files",
        type=["txt", "html"],
        accept_multiple_files=True,
    )

    if uploaded_files is not None:
        file_tokens = {}
        all_tokens = []

        for i, uploaded_file in enumerate(uploaded_files):
            dc = Document()
            tokens = Document.tokenize(dc, uploaded_file)
            file_tokens[uploaded_file.name] = tokens
            all_tokens.extend(tokens)

        search_query = st.text_input("Search a term...")
        if search_query:
            ranker = Rank()
            query_tokens = Document.tokenize(Document(), search_query)
            matching_files = {}

            for file_name, tokens in file_tokens.items():
                total_score = 0
                for query_token in query_tokens:
                    if query_token in tokens:
                        total_score += ranker.tf_idf(query_token, tokens, all_tokens)
                matching_files[file_name] = total_score

            sorted_files = sorted(
                matching_files.items(), key=lambda item: item[1], reverse=True
            )

            if sorted_files:
                st.write("Your term appears in the following files: ")
                for file_name, score in sorted_files:
                    st.write(f"{file_name} (Relevancy Score: {score})")
            else:
                st.write("No matches found :(")


if __name__ == "__main__":
    main()
```

***Figure IV -** The **Main.py** file.*

In the main file, as seen in **Figure IV**, the main function combines all of the aforementioned classes, and uses their functions, alongside the Streamlit framework, to handle and display all the necessary information based on the queries, while also providing some insight for the TF-IDF algorithm's score.