**CSCE 4460: Project Report**

**Project: Code Profiling and Performance Analysis Tool**

**Nil Patel**

**April 22, 2024**

**Professor: Dr. Wajdi Aljedaani**

**Introduction**

The Flask application Code Profiling and Performance Analysis Tool is designed to provide a user-friendly interface for profiling Python code and analyzing its performance metrics. By leveraging libraries such as cProfile, pandas, matplotlib, and seaborn, the application allows users to upload Python code files, profile them, and visualize the profiling results through various statistical analyses and graphical representations.

**Application Architecture:**

The application follows a typical Flask architecture, consisting of routes, functions, and templates:

Routes:

* The index route (`/`) renders the main page where users can upload files and view profiling results.
* The upload route (`/upload`) accepts file uploads, profiles the code, generates statistical analyses and visualizations, and renders the main page with the results.

Important Functions:

* profile\_code: Profiles the uploaded Python code using cProfile and returns the profiling data.
* parse\_profile\_data: Parses the profiling data into DataFrames, separating it based on cumulative time, total time, and function call counts.
* create\_visualization`: Creates bar plots based on the parsed profiling data for cumulative time, total time, and function call counts, and saves them as static files.
* extract\_summary : Calculates various summary statistics from the profiling data, including total cumulative and total time, percentage contributions of functions to total time, and identification of functions with high cumulative and total times and call counts.

**Code Execution Flow**

The execution flow of the application can be broken down into the following steps:

* **File Upload and Handling**:
  + The application expects the user to upload a Python code file via a web interface.
  + It checks if a file has been uploaded and if not, returns an error message.
  + Upon successful upload, it reads the content of the file for further processing.
* **Code Profiling**:
  + The uploaded code is executed using the **exec** function within the **profile\_code** function.
  + During execution, cProfile is utilized to gather profiling data, including information on function calls, cumulative time, and total time.
* **Data Parsing**:
  + The profiling data obtained from cProfile is parsed into structured formats for analysis, specifically pandas DataFrames.
  + Separate DataFrames are created for cumulative time, total time, and function call counts, facilitating individual analysis of these metrics.

**Profiling Data Analysis**

The application performs comprehensive data analysis on the profiling results:

* Cumulative Time: Calculates the cumulative time spent in each function and identifies functions with high cumulative times.
* Total Time: Calculates the total time spent in each function and identifies functions with high total times.
* Function Call Counts: Calculates the number of calls made to each function and identifies functions with high call counts.

**a. Parsing Profiling Data:**

* The **parse\_profile\_data** function is responsible for parsing the raw profiling data into pandas DataFrames.
* It splits the profiling data into lines and further divides each line into columns based on whitespace.
* Separate DataFrames are created for cumulative time, total time, and function call counts, each containing columns such as 'ncalls', 'tottime', 'cumtime', and 'filename:lineno(function)'.

**b. Statistical Analysis:**

* **Cumulative Time Analysis**:
  + The **df\_cumtime** DataFrame contains information on cumulative time for each function.
  + Statistical analysis involves calculating the total cumulative time and identifying functions with high cumulative times (>5% of total time).
* **Total Time Analysis**:
  + Similar to cumulative time analysis, the **df\_tottime** DataFrame contains information on total time for each function.
  + Functions with high total times (>5% of total time) are identified for further analysis.
* **Function Call Counts Analysis**:
  + The **df\_ncalls** DataFrame provides data on the number of calls made to each function.
  + Functions with high call counts (>5% of total calls) are identified.

**c. Visualization:**

* Bar plots are generated using seaborn to visualize the profiling data:
  + **Cumulative Time Plot**: Shows cumulative time for each function, aiding in the identification of functions with significant time consumption.
  + **Total Time Plot**: Displays total time for each function, providing insights into functions contributing the most to overall execution time.
  + **Function Call Counts Plot**: Illustrates the number of calls made to each function, helping identify frequently called functions.

**Conclusion**

The Flask application for code profiling and analysis offers a powerful tool for developers to gain insights into the performance characteristics of their Python code. By combining profiling techniques with statistical analysis and visualization, the application enables users to identify performance bottlenecks, optimize code efficiency, and enhance overall application performance.