Bash Grader Project

Malay Kedia

April 28, 2024

Contents

1 Introduction

In schools and colleges, keeping track of student details with their marks and grades is essential. BASH GRADER is a helpful tool designed to handle CSV files containing this information.
The project aims to streamline various tasks related to CSV file management through a set of commands executed via a bash script named submission.sh. First, it combines different CSV files into one, so all the student information is in a single place. Then, it can calculate totals and statistics related to all exams, helping teachers understand how students are doing overall. It can allow TAs or teachers to update marks in case of changes in marks after crib sessions. It has a linear VCS system built into it, to keep track of changes and updates in the CSV files.
This project aims to simplify the process of managing student data. It's like having a handy assistant to organize and analyze grades effortlessly. In this report, we'll explain how BASH GRADER works, its features, and how it can make life easier for educators.

2 Bash Script Documentation

2.1 Utilities

2.1.1 Combine

The `combine` command merges multiple CSV files containing the marks of students in different exams into a single `main.csv` file. Thus, the output file contains student information such as roll numbers, names, and their respective marks from each input CSV file.
Usage:

- To combine all csv files in current directory:

```
bash submission.sh combine
```

- To combine given csv files in the provided order:

```
bash submission.sh combine <file1.csv> <file2.csv> ...
```

It verifies the existence and format of all provided files; creates the header for `main.csv` with Roll Number, Name and various exam-name columns; appends the names and roll numbers from each provided file to `main.csv`;and finally adds the marks from each provided file to `main.csv`, entering 'a' if the student was absent in a particular exam based on the corresponding roll numbers.

2.1.2 Upload

The `upload` command facilitates the uploading or importing of files to the current directory.
Usage:
To copy files to the current directory:

```
bash submission.sh upload <file1> <file2> ...
```

It verifies the existence of the specified files and then copies them to the current directory. After successful copying, it confirms the same on terminal.

2.1.3 Clean

The `clean` command enables users to remove specified files from the current directory.
Usage:
To remove specified files from the current directory:

```
bash submission.sh clean <file1> <file2> ...
```

It checks for the existence of each specified file and removes them from the current directory if they exist. After successful deletion, it confirms the same on terminal.

2.1.4 Scale

The `scale` utility adjusts the marks of each student in an exam, by scaling all marks by a constant factor. This is useful when the weightage of 1 mark in 2 exams is different, and hence, for totaling, we want to scale the exam marks.
Usage:
To scale in exam `examname` by changing original max marks to final max marks:

```
bash submission.sh scale <examname> <maxmrks_orig><maxmrks_fin>
```

It scales the marks of each student relative to the original maximum marks and the desired final maximum marks. If `main.csv` exists, it updates the same.

2.1.5 Total

The `total` command calculates the total marks of each student based on their individual subject marks.
Usage:
To calculate the total marks of each student:

```
bash submission.sh total
```

It sums up the marks of each student across all subjects, treating 'a' as zeros, and appends the total marks to each student's record in `main.csv`. If `main.csv` doesnt exist, it prompts the user to run the combine command first. Once run, it is always run on its own whenever changes are made to main.csv.

2.1.6 Update

The `update` command allows users to modify the marks of students in a dataset, allowing changes in dataset after cribs, allowing easy and interactive access to dataset.
Usage:
To update the marks of a student:

```
bash submission.sh update
```

On running the command, the user is prompted to specify the exam for which marks are to be updated. Subsequently, the script prompts the user to input the roll number of the student whose marks require modification. If the specified roll number does not exist in the dataset, the user is given the option to add the student, including their name and initial marks. Conversely, if the roll number exists, the script retrieves the corresponding student's name and confirms the selection with the user. After successfully updating the marks, the script offers the option to repeat the process for another student. Once all updates are completed, the updated dataset is consolidated using the bash submission.sh combine command if `main.csv` had existed before running the command.

2.2 Implementation of VCS

The code provides a custom version control system (VCS) similar to Git, offering essential functionalities for managing repositories containing files like CSVs and PNGs. The Git repository structure comprises directories for storing various components such as staged changes, commit history, and working files. In this system, a symbolic link (symlink) is utilized to track the location of the Git repository. This symlink serves as a reference point, enabling the scripts to locate and operate within the repository directory structure seamlessly. By establishing this symbolic connection, the VCS scripts can access and manipulate the repository's contents, including staging changes, committing snapshots, and retrieving historical versions.

2.2.1 Git_init

The `git_init` command initializes a remote repository at the specified location and uses the same for version control.
Usage:
To initialize a remote repository:

```
bash submission.sh git\_init <path\_to\_remote\_repo>
```

It creates the necessary directory structure within the specified path, including directories for storing staged changes and commit history. Additionally, it sets up essential files like `git_log.txt` and `git_files_deleted_from_stage.txt` to track commit history and deleted files from the staging area, respectively. A symbolic link (`.my_git`) is established to reference the remote repository's location, facilitating easy access for subsequent VCS operations.

2.2.2 Git_add_to_stage

The `git_add_to_stage` command is used to add addition, modification or deletion of files to the staging area in preparation for a commit in the version control system.
Usage:

- To add new/modified files to the staging area:

```
bash submission.sh git_add_to_stage <file1> <file2> ...
```

- To add deletion of files to the staging area:

```
bash submission.sh git_add_to_stage --delete <file1> ...
```

It copies the files provided as arguments from working directory to the staging area. If no arguments are provided, it prompts the user to specify the files to be added. Alternatively, it can add files marked for deletion by using the `--delete` flag followed by the list of files to be removed, by adding their names to `git_files_deleted_from_stage.txt`.

2.2.3 Git_remove_from_stage

The `git_remove_from_stage` command is used to remove files from the staging area in the version control system. It can remove newly added or modified files from the staging area or cancel the deletion of files marked for removal. Upon successful addition, the command confirms the files added to the staging area.
Usage:

- To remove new/modified files from the staging area:

```
bash submission.sh git_remove_from_stage <file1> <file2> ...
```

- To remove deletion of a file from the staging area:

```
bash submission.sh git_remove_from_stage --delete <file1> <
    file2> ...
```

It deletes the files given as arguments from the staging area in the remote repository. If no arguments are provided, it prompts the user to specify the files to be removed. Alternatively, it can remove files marked for deletion by using the `--delete` flag followed by the list of files to be removed, by removing their names from `git_files_deleted_from_stage.txt`. Upon successful removal, the command confirms the files removed from the staging area.

2.2.4 Git_status

The `git_status` command is used to view the status of the current repository with respect to the staging area in the version control system. It displays changes to be committed files, changes not staged for commit, and untracked files. The output provides a comprehensive overview of the repository's status, aiding users in tracking modifications and managing files effectively.
Usage:
To view the status of the current repository with respect to the stage:

```
bash submission.sh git_status
```

The command determines the latest commit's hash and then compares the files in the staging area with those in the latest commit to identify new, modified, and deleted files. Additionally, it detects changes not staged for commit and lists untracked files.

2.2.5 Git_commit

The `git_commit` command is used to commit changes in the staging area to the remote repository in the version control system. It allows users to commit with or without a message.
Usage:

- To commit the stage to the remote repository:

  ```
  bash submission.sh git_commit
  ```

- To commit the stage to the remote repository with a message:

  ```
  bash submission.sh git_commit -m <message>
  ```

It checks for files staged for commit. If no files are staged, it exits with a message indicating there are no files to commit. It then prompts the user to enter a commit message if one is not provided as an argument. The command creates a commit with a unique hash and appends the commit details to the `git_log.txt`. It copies all files from previous commit not in `git_files_deleted_from_stage.txt` and overwrites them with staged files to the commits directory. Additionally, displays the changed files between the current and previous commits. Finally, it clears the staging area after a successful commit.

2.2.6 Git_log

The `git_log` command is used to view the log of commits in the remote repository in the version control system.
Usage:
To view the log of commits:

```
bash submission.sh git_log
```

The command displays the content of the `git_log.txt` file, which contains the commit history with details such as commit hash, timestamp, and commit message.

2.2.7 Git_checkout

The `git_checkout` command allows users to switch to a specific commit in the version control system. It supports three different methods of checkout: by commit hash, by specifying the number of commits before HEAD, and by commit message.

Usage:

- To checkout the commit n commits before HEAD:

```
    bash submission.sh git_checkout HEAD[~n]
```

- To checkout a commit by hash:

```
    bash submission.sh git_checkout <hash>
```

- To checkout a commit by message:

```
    bash submission.sh git_checkout -m <message>
```

The command checks out the specified commit by copying and hence overwriting the files from that commit to the current working directory. If multiple commits are found with the same message, it prompts the user to provide a more specific identifier.

2.3 Analytics

In this section, we delve into various commands related to analytics aimed at analyzing and visualizing data. The scripts include a cover a range of analytical techniques, including calculating statistical measures and the analysis of relationships between variables through correlation analysis. Visualizations are a powerful tool for understanding data patterns, and the commands facilitate the creation of histograms and scatter plots, respectively, to visualize distributions and relationships within the data. Finally, there is functionality for assigning grades. Through these analytics scripts, we aim to provide a comprehensive toolkit for exploring and understanding student data.

2.3.1 Closest_name

The `closest_name` command is designed to find the closest name to a given input name. It provides a useful utility for tasks such as name matching or fuzzy string matching. It takes a name as input and returns the closest matching name from `main.csv`. It can be particularly useful in scenarios where approximate matching or correction of input data is required.

Usage:

To find the closest name to a given name:

```
bash submission.sh closest_name <name>
```

The command employs metric of levenshtein distance to find the string closest to the given name among the database, by returning the string from the database which had the lowest levenshtein distance from our input string. The implementation of levenshtein distance uses only python and no other libraries.

2.3.2 Student_performance

The `student_performance` command is a tool for accessing and printing the marks of students based on their roll numbers or names. It offers flexibility by allowing users to search for marks using either the student's roll number or name. Additionally, users can choose to print the marks of the closest name or roll number to the provided input, which can be beneficial for handling minor input discrepancies. If specified, the `--graph` option plots the student's marks relative to the maximum and average marks obtained in each exam.
Usage:

- To print the marks of a student from roll no.:

  ```
      bash submission.sh student_performance <student_roll_no>
  ```

- To print the marks of a student from name:

  ```
      bash submission.sh student_performance --name <
          student_name>
  ```

Options:

- `--close`: Specify to print the marks of the closest name/roll_no to the given name.

- `--graph`: Display the graph of student performance compared to class average and maximum marks.

It extracts information from `main.csv` to retrieve and print the marks corresponding to the provided input in form of name or roll no., facilitating quick access to student performance data. It utilizes levenshtein distance when given the tag of `--close` to handle minor discrepancies in input names, ensuring robustness and ease of use.

2.3.3 Calc_stats

The `calc_stats` command calculates various statistics for the marks obtained in different exams. It accepts exam names as arguments and computes statistics such as the total number of students present, maximum score, minimum score, mean, standard deviation, skewness, quartiles, and mode. The script

supports analyzing multiple exams simultaneously, providing insights into the performance distribution across different assessments.
Usage:
To calculate the statistics of the marks for specific exams:

```
bash submission.sh calc_stats <examname1> <examname2> ...
```

The command utilizes NumPy and SciPy libraries for efficient numerical computation and statistical analysis. If no exam names are provided, the script calculates statistics for all available exams by default. Additionally, specifying ”total” as an exam name calculates statistics for the total marks, requiring prior execution of the ‘total‘ command to generate the necessary data. Finally, the results are printed to the console.

2.3.4 Calc_correlation

The `calc_correlation` script calculates the correlation between the marks obtained in different exams. It accepts two exam names as arguments and computes the correlation coefficient between them. Additionally, it provides an option to print the correlation matrix for multiple columns, offering insights into the relationships among various assessments.
Usage:

- To calculate the correlation between the marks of two exams:

  ```
      bash submission.sh calc_correlation <exam1> <exam2>
  ```

- To print correlation matrix for multiple columns:

  ```
      bash submission.sh calc_correlation --matrix <exam1> <
          exam2> ...
  ```

The script utilizes NumPy for efficient numerical computation and calculates the correlation coefficient using Pearson correlation.

2.3.5 Plot_histogram

The `plot_histogram` command generates a histogram to visualize the distribution of marks for a specific exam. It provides options to customize the plot by setting the maximum and minimum values for the marks, specifying the output file for the plot, and adjusting the number of bins in the histogram.
Usage:
To plot a histogram of marks of an exam:

```
bash submission.sh plot_histogram [options] examname
```

Options:

- `--maxmarks <value>`: Set the maximum value for the marks.

- `--minmarks <value>`: Set the minimum value for the marks.
- `-o <output_file>`: Specify the output file for the generated plot.
- `--bins <value>`: Set the number of bins in the histogram (default: 10).

The script utilizes Matplotlib to create the histogram plot, providing a visual representation of the mark distribution.

2.3.6 Plot_scatter

The `plot_scatter` script generates a scatter plot to visualize the correlation between marks of two different exams. It provides an effective means to explore potential correlations or patterns between exam performances. It also allows users to specify the output file for the generated plot.
Usage:
To plot a scatter plot of marks of two exams:

```
bash submission.sh plot_scatter [options] exam1 exam2
```

Options:

- `-o <output_file>`: Specify the output file for the generated plot.

The script utilizes Matplotlib to create the scatter plot, visualizing the relationship between marks obtained in two distinct exams.

2.3.7 Grade

The `grade` command is used to assign grades to students based on their marks. It allows for flexible grading approaches, including relative grading based on mean and standard deviation, absolute grading with custom grade boundaries, and clustering-based grading. It provides various options for grading methods and customization of grade baskets.
Usage:
To create a CSV file containing the grades of the students, along with their marks used for assigning grades:

```
bash submission.sh grade <output_file> [options]
```

Options:

- `--baskets`: Specify custom grade baskets in descending order. Default is ['AA', 'AB', 'BB', 'BC', 'CC', 'CD', 'DD', 'FF'].
- `--clustering`: Use clustering-based grading.
- `--relative`: Use relative grading based on mean and standard deviation.
- `--absolute`: Use absolute grading with custom grade boundaries.

- `--boundaries`: Specify custom grade boundaries when using absolute grading.
- `--criteria`: Specify the criteria for grading (Default is total).

Using NumPy and SciPy modules, it enables users to specify grading preferences such as relative, absolute, or clustering-based grading, along with options to define custom grade baskets and boundaries. Leveraging statistical computations like mean, standard deviation, and clustering algorithms like K-means, the script efficiently assigns grades to students, ensuring fair and accurate assessment. The output is written to the specified output file, including student roll numbers, names, marks, and corresponding grades.

2.3.8 Report_Card

The `report_card` command generates a comprehensive report card for a student, presenting detailed information about their academic performance. It allows users to specify the student's roll number and provides options for customization such as specifying the output file name and providing a separate file containing grades. Additionally, the generated report cards are saved in the "Reports" directory for easy access and organization.

Usage:

To generate the report card of a student:

```
bash submission.sh report_card <student_roll_no> [options]
```

Options:

- `-o <output_file>`: Specify the output file for the generated report card (default: report_roll_no.pdf).
- `--grades_file <file>`: Specify the file containing the grades of the students.

Utilizing Python's data processing capabilities along with LaTeX for document generation, the script retrieves student performance data from the main CSV file and creates a visually appealing report card. It offers flexibility in specifying output file names and incorporating external grade data files for accurate grading. The report cards are organized and stored in the "Reports" directory, ensuring easy access and management. The command enhances the process of generating student report cards, providing a convenient and efficient solution for academic assessment.

3 Conditions for the Script to Work

Certain inherent assumptions are made in the functioning of the script:

- The script is designed to run on an Ubuntu operating system. While it may work on other operating systems, it has primarily been developed and tested on Ubuntu.
- The script relies on GNU Awk, Sed, and Python 3 being available and accessible on the system. Additionally, it requires certain Python libraries such as NumPy, Matplotlib, and SciPy to be installed and properly functioning.
- Proper setup and configuration of pdflatex is necessary for the generation of PDF files. The script assumes pdflatex is installed and operational.
- File names or paths should not contain spaces. While some commands may have built-in support for handling spaces in file names, others may rely on parsing strings based on spaces, leading to potential errors if spaces are present.
- CSV files should not have random white-spaces, or empty lines at the end.
- It is advisable to not name the output file of command `bash submission.sh grade filename` with a ".csv" extension, as the `combine` command, whenever run afterwards without specifying the arguments will consider it as a CSV file and attempt to combine its third column with other CSV files.
- Temporary files are created and removed during the execution of the commands. These temporary files are named arbitrarily, such as `temp1212.csv`. Therefore, it is advisable not to use filenames matching this pattern in the working directory. Similarly, the filename `tempPerf.png` should not be used for any image files to prevent conflicts.

4 Error Handling

Error handling is crucial in maintaining the reliability and robustness of any script or utility. This report provides an overview of how error handling is managed across different utilities, Git operations, and analytics scripts.

4.1 Submission.sh

It is ensured that the script is executed from its intended directory by comparing the current directory with the directory containing the script, displaying an error message if they dont match. Additionally, an error is returned if an invalid argument is provided to `submission.sh`

4.2 Utilities

Each utility script begins by checking the validity of the input arguments provided by the user. If the arguments are insufficient, invalid, or if the specified

files are not found, informative error messages are displayed, guiding the user on the correct usage or necessary actions to resolve the issue.
The scripts related to bringing changes in `main.csv` (eg. total) show an error when run before running of combine itself.
Any commands (eg. update, scale) which change the contents of CSV file, automatically update the `main.csv` file if the data of that exam was present in it.
Any commands which remake `main.csv` automatically run `total` if the preexisting `main.csv` had a total column.

4.3 Operations of VCS

All git operation scripts, except `git_init` validate the execution environment by ensuring that the symlink points towards the remote repository, and that the repository has been initialized before proceeding.
Changes in staging area are committed only after checking if the supplied filenames exist in working directory for modifications and additions, and if the filenames exist in last commit but not in the working directory for deletions.
Detailed error messages guide users on the necessary steps to rectify the situation, such as running initialization scripts (`git_init`) or performing prerequisite actions (`combine`) before executing certain commands.

4.4 Analytics

All these commands are implemented using python, which has easier handling of flags and data-analysis. The command is parsed for existence of flags, and checked if any extra undefined flags or arguments are given, which return an error.
They validate input arguments, check for the existence of required data files, and ensure proper initialization before executing complex statistical computations or grading algorithms. Informative error messages are provided to guide users on resolving issues, such as specifying correct input files, ensuring proper data formatting, or running prerequisite scripts (`total`) for necessary data aggregation.

4.5 Conclusion

Overall, robust error handling mechanisms are integrated into the script to enhance user experience, minimize unexpected failures, and provide clear guidance in resolving issues. By proactively addressing potential errors and guiding users through corrective actions, these scripts promote smoother execution and facilitate efficient data analysis, version control, and utility operations within diverse computing environments.

5 Code Modularity

Modularization is a crucial aspect of software development, promoting code organization, reusability, and maintainability. In the provided scripts, modularity is achieved through the division of code into functions and the segregation of functionality into separate files.

5.1 File-based Modularity

The code for each command has been written in separate files, appropriately named and organised according to category in the `Scripts` folder. These files are run whenever a suitable command is passed to submission.sh. Hence, the functionality is distributed across multiple files, with each file focusing on a specific aspect of the overall task.

5.2 Function-based Modularity

The scripts themselves consist of multiple functions, each responsible for a specific task. This modular approach makes it easier to implement changes to tasks performed repeatedly, as modifications need to be made only in one place. Additionally, the `utilities` and `git_functions` directories each contain an additional file called `additional_functions.sh`. These files are imported into all scripts and contain functions used across multiple scripts. Therefore, this function-based modularity promotes code reuse and simplifies testing and troubleshooting.

5.3 Conclusion

By adopting both function-based and file-based modularity, the codebase is structured in a way that promotes organization, extensibility, and maintainability. Developers can easily locate and modify specific functionality without impacting other parts of the system. Additionally, this modular approach fosters code reuse, allowing for the integration of common functions across different scripts and projects.

6 Scope for Further Improvements

While the current set of scripts provides a solid foundation for managing tasks related to utilities, Git operations, and analytics, there are several areas for potential enhancement.

- optimizing the code for performance by implementing efficient algorithms and data structures can enhance execution speed, especially for larger datasets in `combine` command and analytics commands.

- optimizing the version control system using diff and patch techniques can enhance its efficiency in managing changes to files. By only storing the differences between versions, storage space and bandwidth usage can be minimized.

- integrating a branched version control system into the Git operations can provide users with more flexibility and control over their versioning workflows. This can include features such as branching, merging, and conflict resolution to support maintaining remote repositories across multiple devices.

- adding a command to generate a report card based on the graded student data can streamline the process of creating and sharing performance reports.

- expanding the functionality to support more advanced analytics techniques and visualization options could provide teachers with more comprehensive insights into the data of their class.

- ensuring compatibility with a wider range of operating systems and environments can make the scripts more versatile and accessible to a broader audience.

Overall, these potential improvements can contribute to making the scripts more powerful, efficient, and user-friendly for various use cases and scenarios.

7 Conclusion

In conclusion, the implementation of various utilities, Git functionalities, and analytics scripts demonstrates a systematic approach to automating common tasks and managing version control efficiently. The use of modular code design fosters reusability, simplifies maintenance, and enhances the scalability of the codebase. Through error handling mechanisms and structured organization, the scripts ensure robustness and reliability in handling diverse scenarios. Overall, this report showcases a comprehensive framework for streamlining development workflows, promoting collaboration, and empowering users with tools for effective project management and data analysis.

8 References

- ChatGPT

- GitHub Copilot

- Resources provided for CS108 (Slides, Example files, Tutorial files)