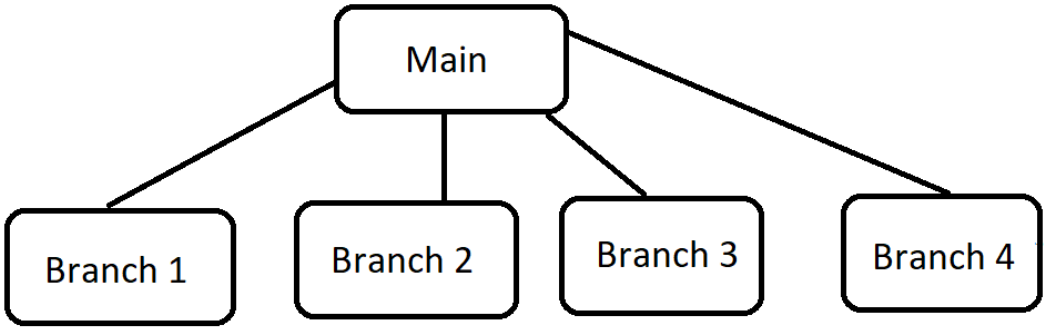
IDS 594 Machine Learning Deployment Project

Deploying a classification problem with multiple models using GitHub Desktop (without the use of command prompt). GitHub - Link

Pragya Arora & Ainesh Panda

Introduction to GitHub Desktop

GitHub Desktop is a free, open-source application that helps you to work with files hosted on GitHub or other Git hosting services. Like any tool for contributing changes to repositories on GitHub, GitHub Desktop is built around the version control software Git. Because GitHub Desktop has a graphical user interface, it simplifies many of the aspects of Git that can be challenging for new users, such as memorizing commands and visualizing the changes you're making. Because GitHub Desktop makes commands like these easy to find, and helps you visualize the changes you're introducing with an integrated diff view, it encourages best practices and helps you to create an accurate and easy-to-follow commit history so other collaborators on a project can easily review your work. Tools such as VS Code, Spring Boot and many more use more or less the same functionality to execute git commands from the UI, and GitHub gives that experience that can be extremely beneficial to work with when handling different tools in a company server environment. To give some context on the functionality of GitHub, one needs to understand the how a normal Git in general works.



The diagram above shows how branching works. We can see a main branch dividing like a tree into multiple branches. One can work in the branch and by merging the changes to the main branch, to modify the source document.

Dataset

The data- set obtained from the Rice_MSC_Dataset.csv used for this project was obtained through Kaggle from Muratkoklu data repository [DATASETS]. The dataset contains spectral data obtained from NIR analysis of 252 rice samples, along with various attributes such as grain variety, growth conditions, and quality ratings. The spectral data was converted into a dataset with 5 classes of rice and the same has been consumed by us in this project.

We started by checking the dataset characteristics such as the number of instances, number of attributes, and any missing values. The shape of the data consists of 75000 rows and 107 columns. We also checked for any missing values in the dataset, and fortunately, there were no missing values.

Information	Value
Dataset Characteristics	Rice MSC Dataset
Attribute Characteristics	Numerical, Multivariate
Associated Tasks	Classification, Regression
Number of Instances	75000
Number of Attributes	107
Missing Values	No
Area	Agriculture
Date of Download	April 7, 2023

Goal and possible solutions

In this project we aim to improve the accuracy and efficiency of rice classification using machine learning algorithms. We intend to publish these models on a public repository such as GitHub using GitHub Desktop for consumption by other users. We aim to explore various functionalities in GitHub while deploying our project on the platform. In addition to the technical aspects, we also intend to tackle common issues arising while operating on the repository. After pulling in our data to the main branch of our local repository we intend to deploy different machine learning algorithms using separate branches. The algorithms we will be testing for rice classification are as follows: Decision Tree, Random Forest, K-means clustering, and Naive Bayes. The algorithms will be tested on the dataset, and the best-performing models will be selected to be merged with the main branch. While working in a team there will be conflicts when we try to make changes to the same repository. In this project, we intend to tackle such problems with the use of various functionalities within GitHub.

Related work

The topics of GitHub desktop were sourced from various articles and videos. Below are the topics reviewed:

"*Introducing GitHub: A Non-Technical Guide*" by Brent Beer. From this textbook, the basic installation procedure of GitHub and its functionalities were sourced. We also explored the use of Atom, an open-source text editor which has Git and GitHub functionality built into it.

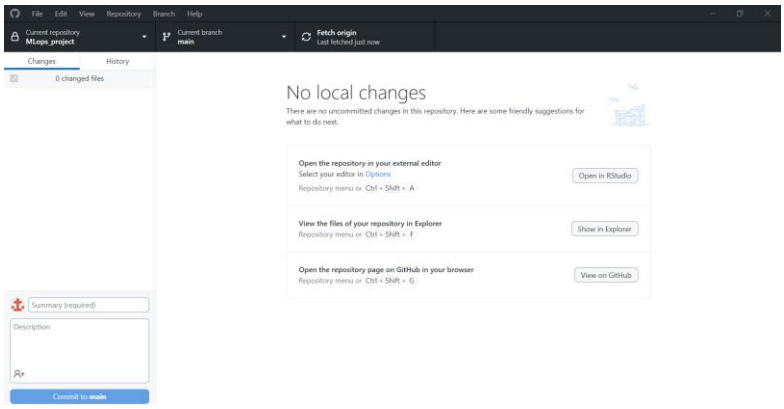
"An Introduction to Version Control Using GitHub Desktop" by Daniel van Strien, which gives a detailed overview of implementing version control in GitHub Desktop

Documentation from GitHub Docs, which gives a quickstart on various Git commands from a desktop, such as push, pull, and cloning repositories along with the visual representation of the actions

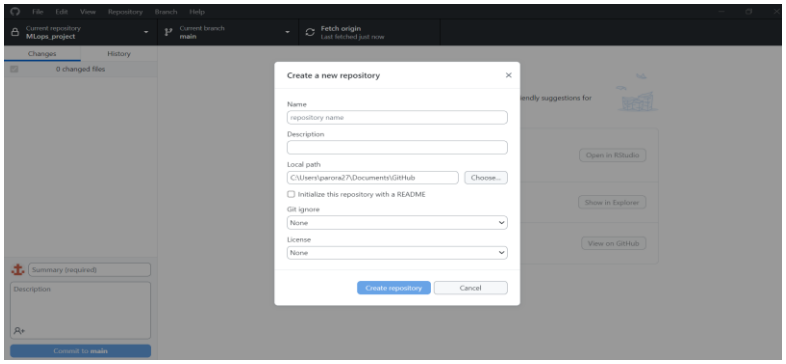
Getting started with GitHub Desktop

In this section we will explore multiple functionalities within GitHub Desktop to understand how the desktop version simplifies the operations performed on command prompt.

1. **Creating a Repository**: Repository can be a great tool to save your work in a consolidated format.

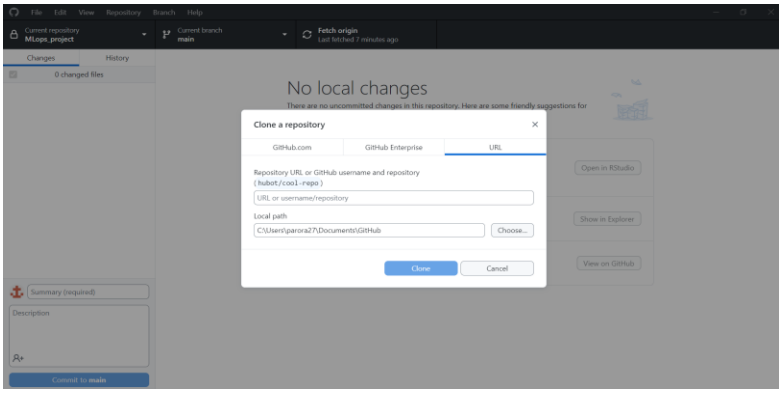


Here is the main screen that we see after opening GitHub desktop. When we click on the file, we see an option to create a new repository. After selecting New repository, we get the following form:



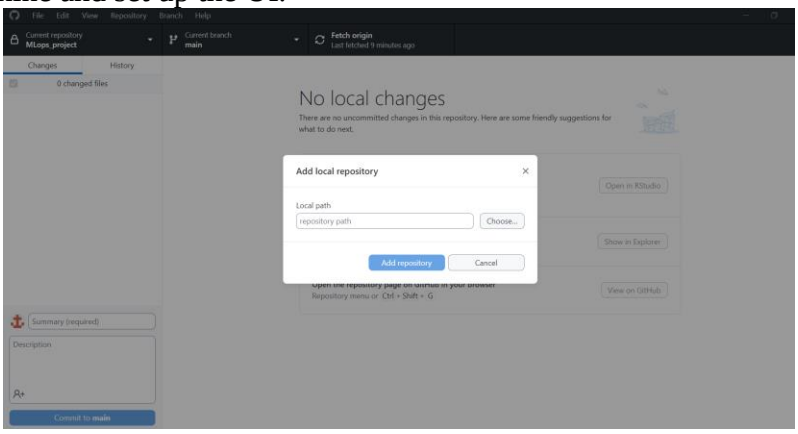
Here we can fill up the information to setup our repository. Now we can make changes on our machine and execute those changes directly in the repository. We can set the path to our repository, ignore any type of files we do not want to commit to our repository, and set a license to manage and share code and materials.

2. **Cloning Repository**: In situations, where we are collaborating with someone in the same GitHub repository, we can simply clone the repository. One may not need to create a new repository and later worry about how to merge the two repos. This is an excellent way to collaborate among a bit number of people who are contributing towards the same project, as in our case, where we are considering a scenario, where multiple people have been working on different models based on the same dataset.

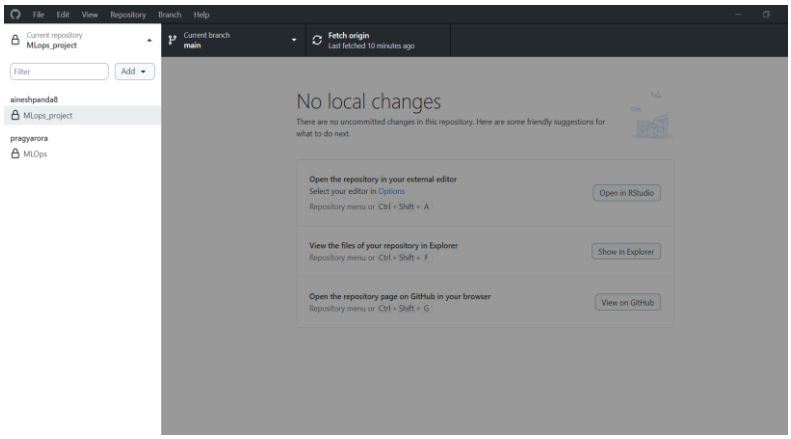


The option to clone repository is also available under the file option.

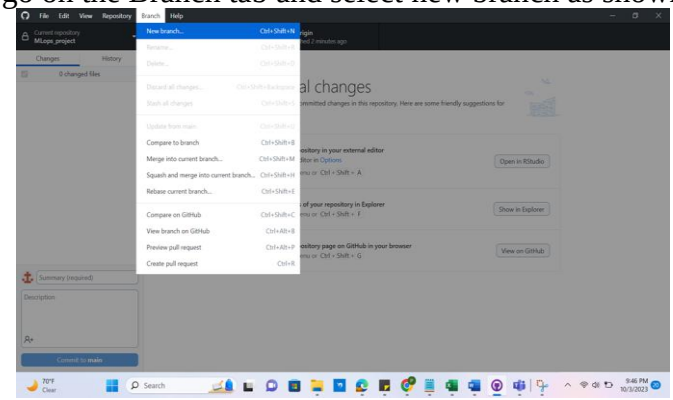
3. **Add Local Repository**: In scenarios where we already have downloaded/cloned a repository in the past, but haven't set it up in GitHub desktop or any other tool such as VS Code, Add repository becomes a very handy tool as it helps us navigate the repository in our local machine and set up the UI.



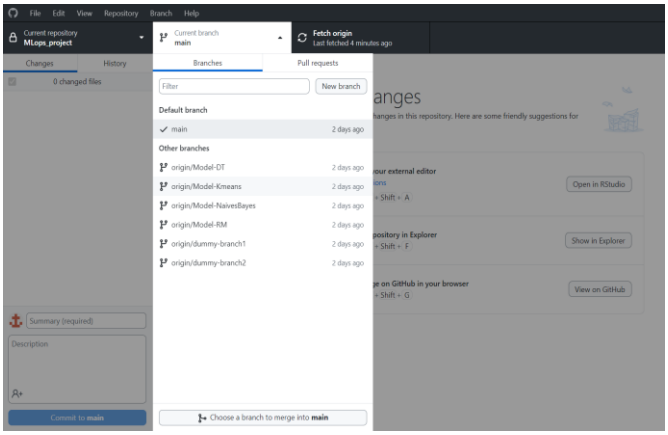
This option is also available under the File option. We can also do the mentioned three operations, by clicking on the current repository option, and clicking on the Add dropdown button.



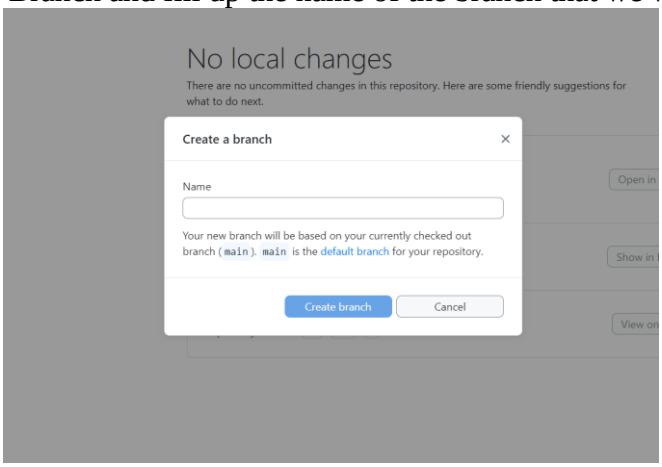
4. **Creating Branches**: There are multiple ways to create Branches in the GitHub desktop. The simplest is to go on the Branch tab and select new branch as shown below:



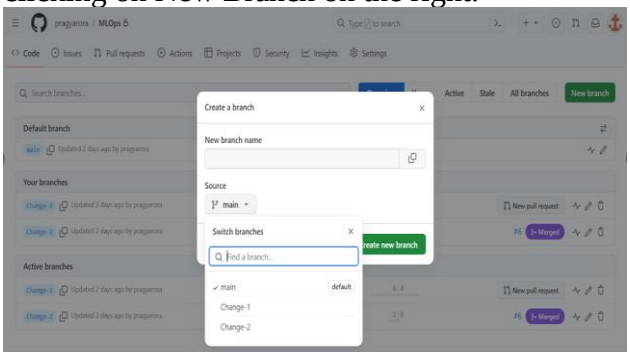
But more common way is to click on the current branch option and select the Current branch tab from the bar under and perform all the brach related operations.



We can select New Branch and fill up the name of the branch that we want.

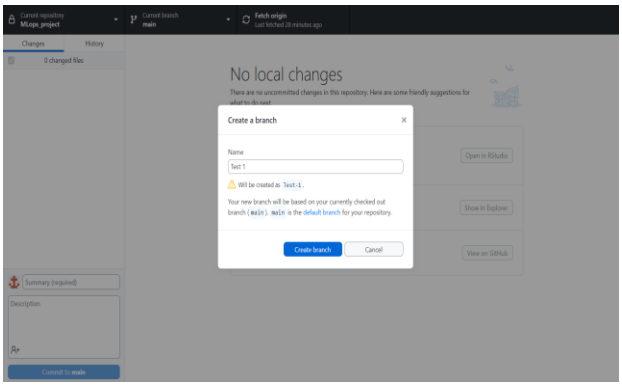


As we can see, the main branch is here considered as the default branch to subbranch will be created with all the information from the main branch, which will be accessible to us to make changes locally without affecting the source, i.e. Main Branch. We can change the default branch setting to another branch from which we want to fetch data locally, but for those operations, are more user-friendly when we do it on GitHub itself, by selecting the Branch tab and then clicking on New Branch on the right.

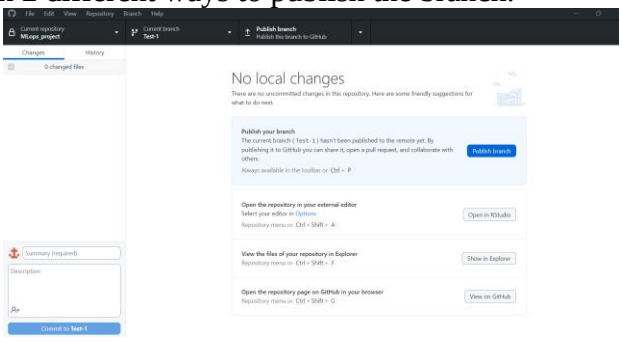


As we can see, we can simply select the branch that we want as our source to create a branch.

5. **Publishing a Branch**:

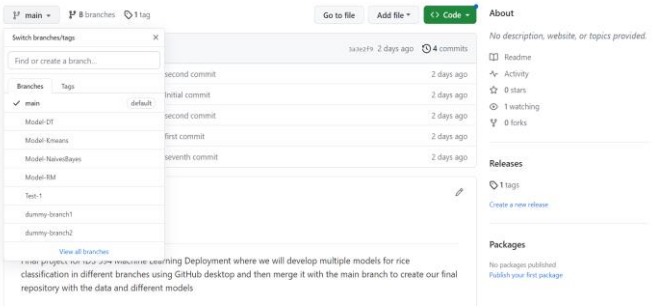


As we see, it automatically shows how the name should be. We don't necessarily have to remember the correct way to name. As soon as we click on Create Branch, we get the following screen with 2 different ways to publish the branch.

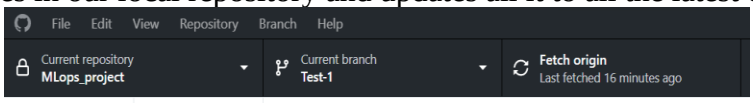


Publishing the Branch means to push the branch to the GitHub repository from our Local Machine.

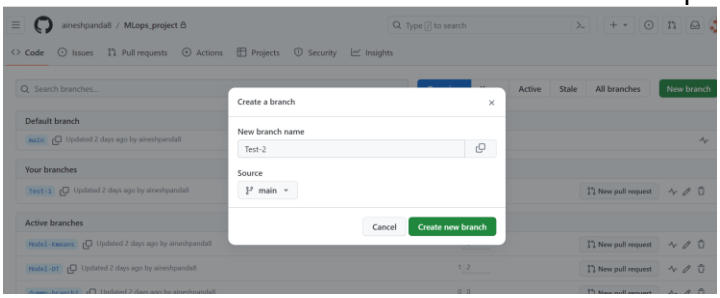
Clicking on Publish shows us an option to create a pull request. Pull request is nothing but a way to apply changes that we will be making to the code base in the Test-1 branch. That is not something that is required at the moment. To verify that all our changes have been pushed successfully, I will go ahead and verify the same at the GitHub repository. We can see the Test-1 branch in the GitHub repository now.



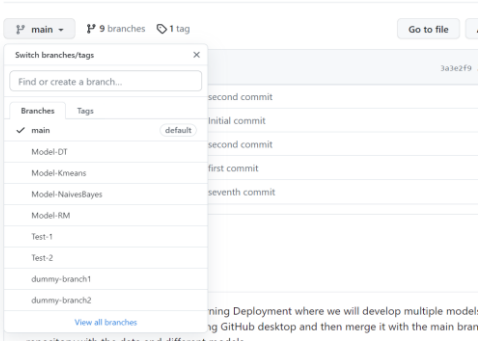
In the GitHub desktop, we can fetch all the information that was updated in the GitHub repository using the Fetch Origin option. When we click on fetch Origin, it basically refreshes the files in our local repository and updates all it to all the latest changes.



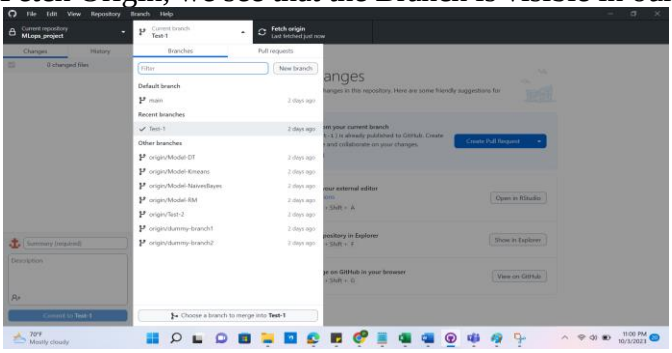
Let’s create a branch directly in the GitHub repository, and try to observe what actions might be needed in order to receive that information in our Git Desktop.



We see the branch on GitHub.

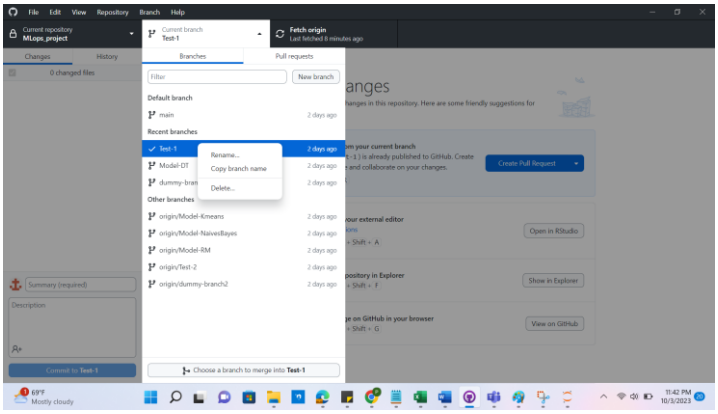


When we click on Fetch Origin, we see that the Branch is visible in our Local repository.

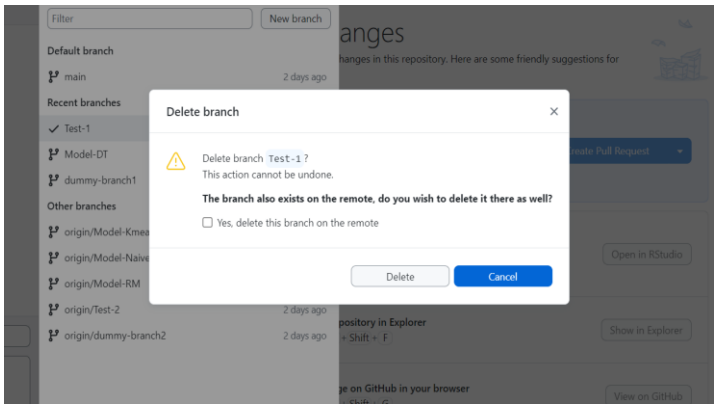


This shows the seamless nature of how GitHub desktop works. The same can also be done using the command line but here we just have to click a button to fetch the information.

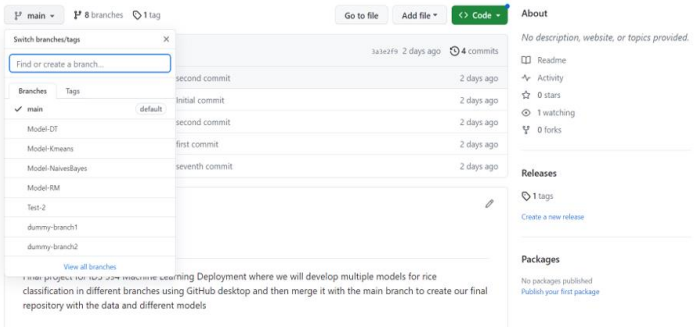
6. **Deleting a Branch**: We can delete a created branch from the GitHub Desktop by simply right-clicking on the branch name.



This is the easiest way to tackle delete.

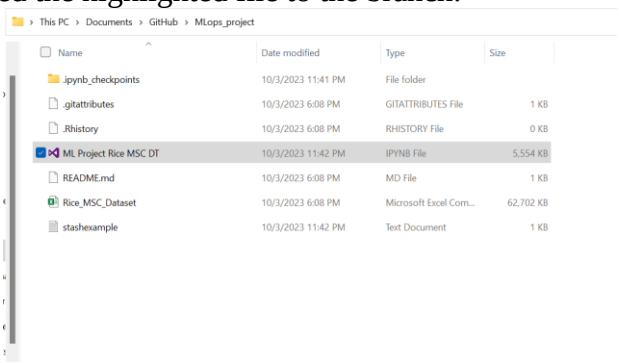


We should also check the box if we want to delete the file from the GitHub repository along with the local repository.

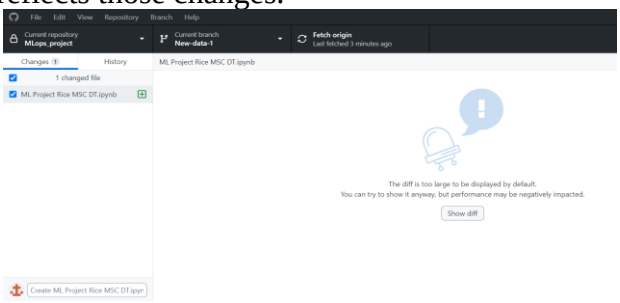


We see that the branch is not available anymore in the remote repository as well.

7. **Pushing Changes in Branch**: For the demo, we have created another branch from dummy-branch-1. I have added the highlighted file to the branch.

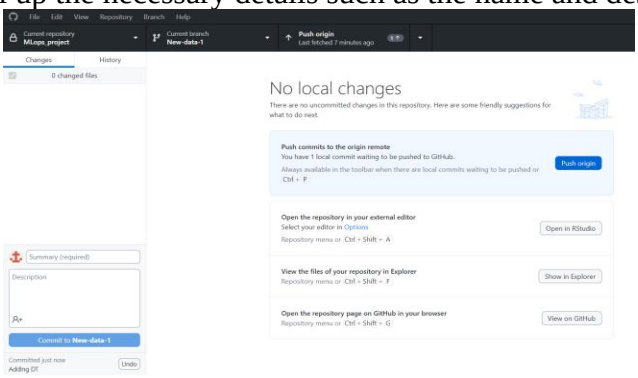


We see, that in Git Desktop, it automatically detects the change that has happened in the local repository and reflects those changes.

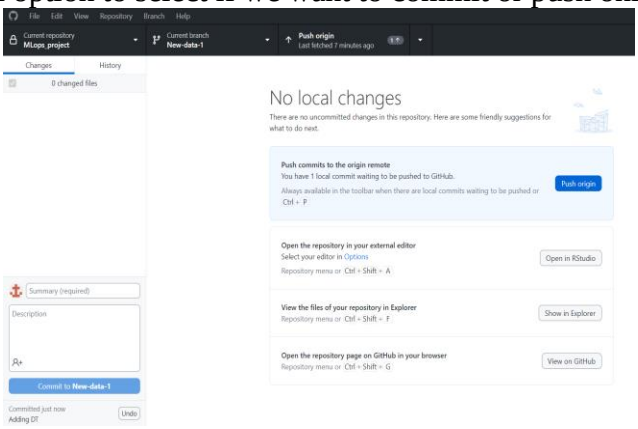


This change can be as simple as changing a few lines of code or adding a brand-new file. We can add a small description to make our commitment more meaningful. In version

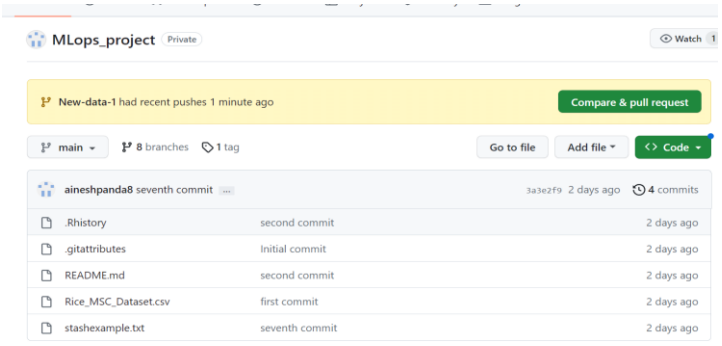
control systems like Git or GitHub Desktop, a "commit" is a pivotal operation. We can encapsulate a defined set of code or file addition or alteration through commits. It provides a snapshot that represents the state of a project at a particular moment in its history and provides a unique point in the timeline of a repository. Committing serves several crucial purposes. First, it maintains a comprehensive version history of the project, enabling developers to reference past states, compare different iterations, and revert to prior versions if necessary. Secondly, commits facilitates collaboration among multiple team members by allowing them to work on distinct changes independently. Commits also play a pivotal role in bug tracking and enhancing code review processes.
Once a change has been made in the repository it appears on the changes section of GitHub Desktop. We can fill up the necessary details such as the name and description.



We immediately see the option to initiate push on the top with the exact counter of the number of changes made in that following branch of the repository. In case of multiple changes, we have an option to select if we want to commit or push only selective changes.

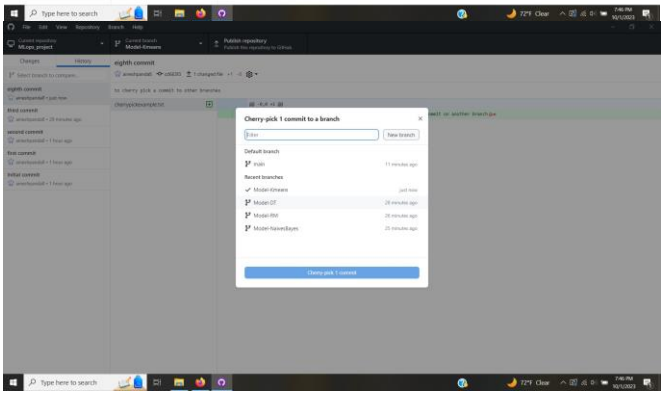


After pushing, we see all the information is updated locally and now we can initiate a pull request to merge in the source branch.

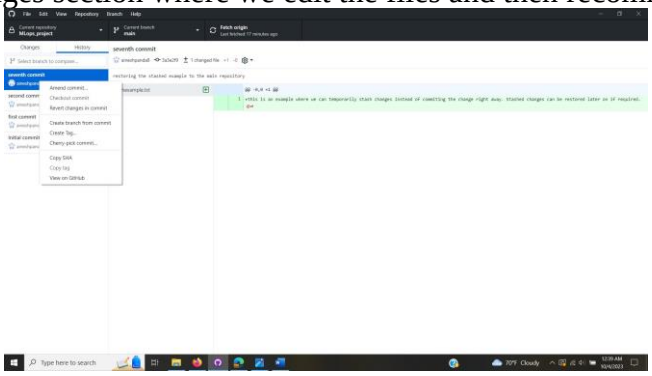


We followed the same steps to create another branch called “New-Data-2” with information on KMeans.

8. **Cherry-picking a commit**: Cherry-picking a commit in Git refers to the process of selecting and applying a specific commit from one branch to another. This allows us to choose individual changes from one branch and apply them to another branch without merging the entire branch. In GitHub Desktop we can right-click on the commit we want to cherry-pick and then choose the destination branch we want our commit to be replicated in. But this action can only be done to one branch at a time.

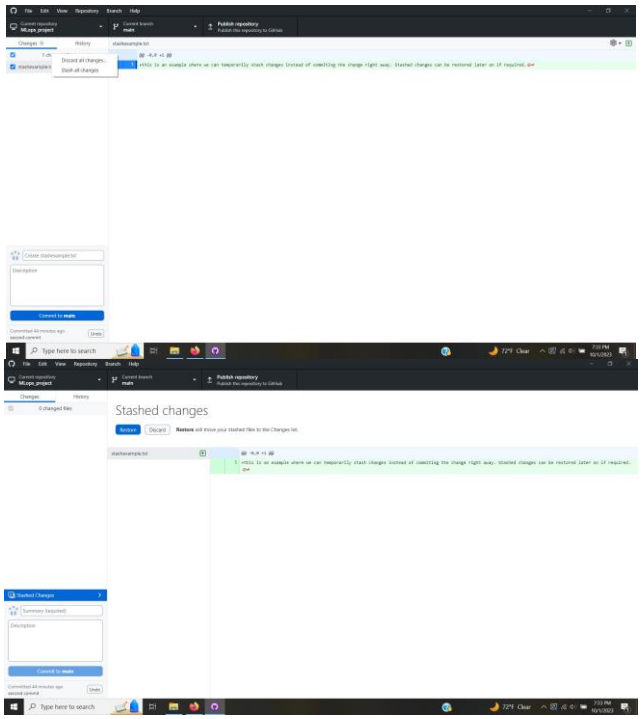


9. **Amending a commit**: Amending a commit in Git refers to the process of making additional changes or corrections to the most recent commit in your project's history. This allows us to refine the contents of the last commit without creating a new commit altogether. This can be done on GitHub Desktop by right-clicking on the most recent commit. This will bring us to the changes section where we edit the files and then recommit it to the branch.

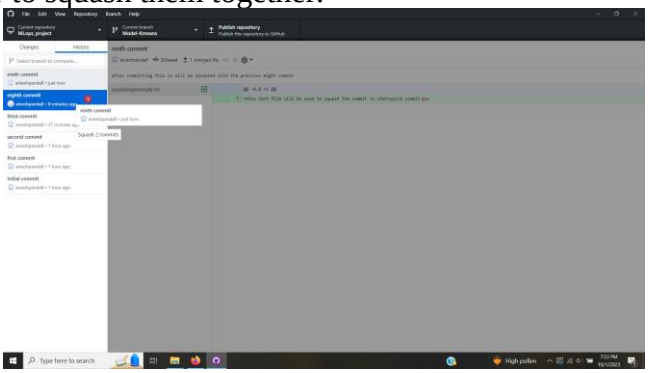


10. **Stashing changes prior to commit**: Stashing is a way to temporarily save your uncommitted changes so that you can switch to a different branch, pull in updates, or

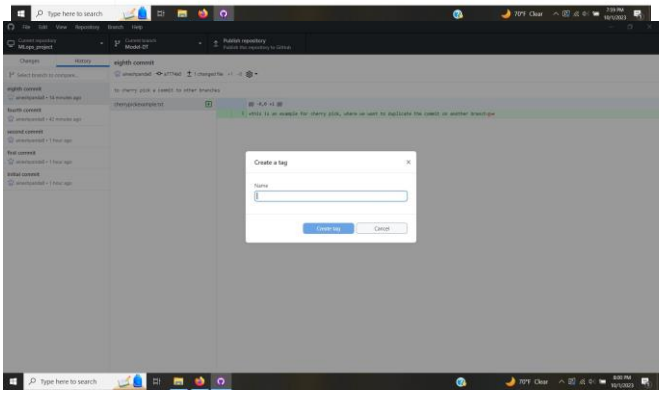
perform other tasks without committing your changes. On the GitHub desktop, all the changes can be selected together and stashed with a right click. When required these stashed changes can be restored to be committed to the branch.



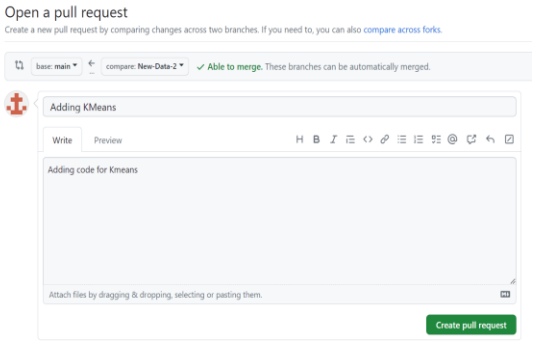
11. **Squashing commits**: Squashing commits in Git refers to the process of combining multiple consecutive commits into a single commit with a consolidated and more informative commit message. Squashing is often used to clean up a branch's commit history, making it more readable and logical, especially before merging a feature branch into the main branch or sharing changes with collaborators. In GitHub Desktop we can drag multiple commits on top of each other to squash them together.



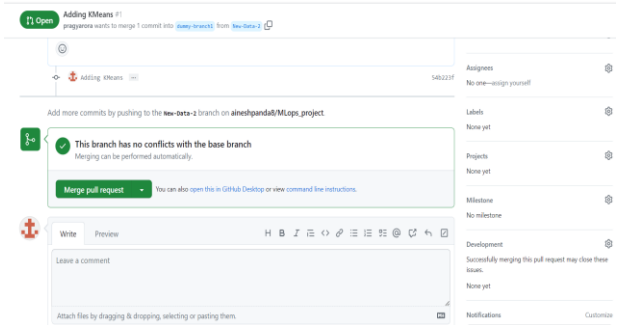
12. **Tagging a commit**: Tagging a commit in GitHub Desktop refers to the process of associating a specific version of your code with a meaningful label, known as a "tag." Tags are typically used to mark significant milestones or releases in your project's development history. They make it easier to reference and identify specific points in your repository's history. On GitHub Desktop a commit can be tagged by a right click and creating a tag which pops up a window to note down appropriate information.



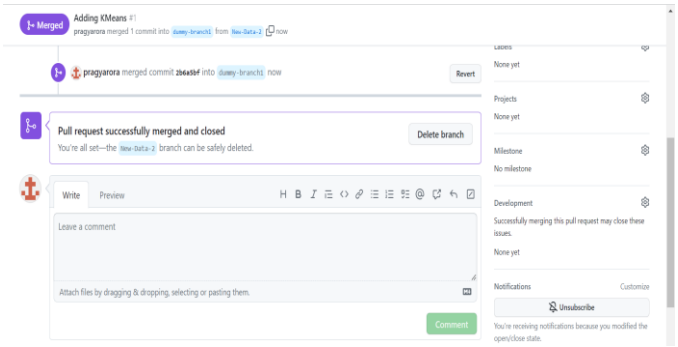
13. **Pull and Merge**: To merge the changes in the dummy-branch-1, we create a pull request for KMeans



After Creating a Pull request, we are able to see if there are any merge conflicts. Merge Conflicts is a concept to understand if the proposed merge of branches is supported by the source branch(can be any branch that we want to merge into).

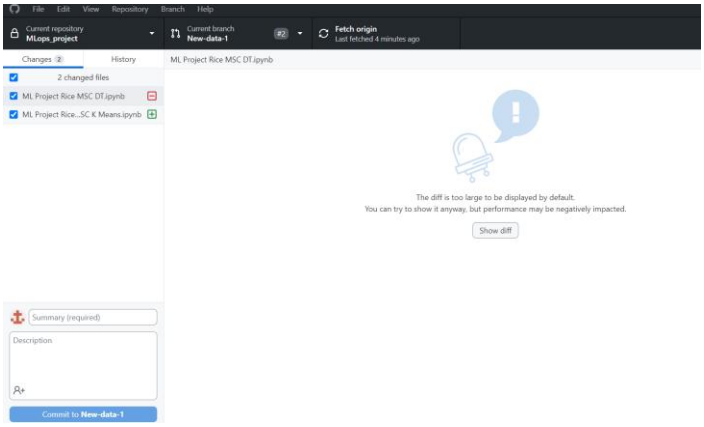


After confirming the merge, we can see that the merge request is completed and closed automatically.



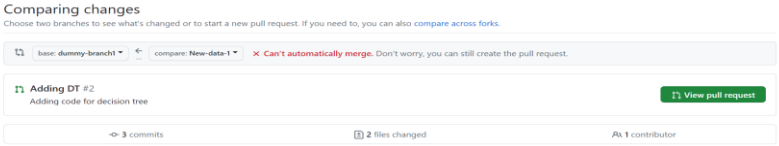
Let's try to execute the same thing for "New-data-1".

14. **Creating Merge Conflict**: Merge Conflict is a very common Scenario in the world of Deployment. To execute the following scenario, I will first update the data in New-Data-2 with the KMeans code with slight variations. We push the new changes to the origin and then make some modifications with the KMeans code.



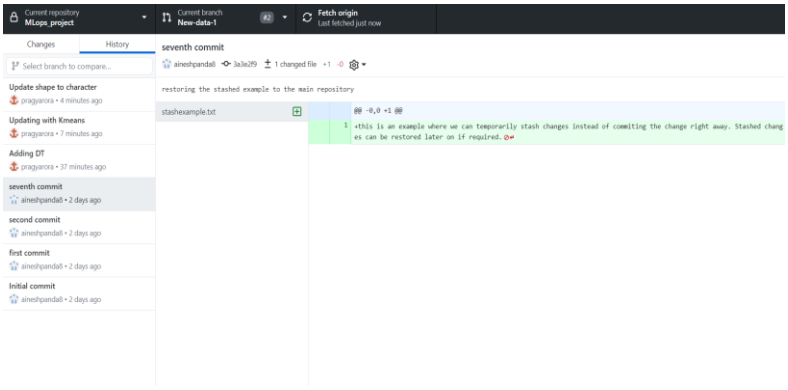
We updated df.shape to df.character

Now when we review the pull request, we see that there are some issues with the merge.

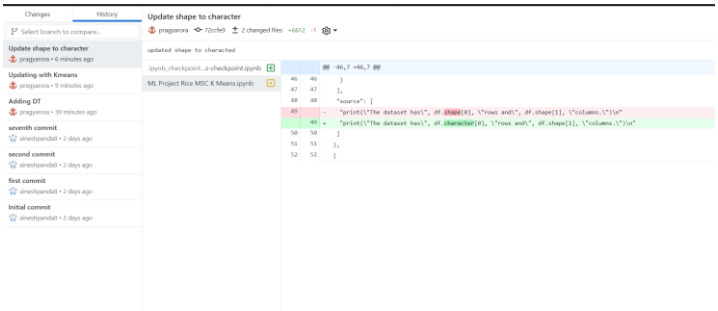


We can review the changes on GitHub.
In case of a merge conflict, one needs to take a fresh pull of the latest version of the source branch and discuss with fellow workers what is needed to be done. Whether the new changes that you have made will be done on top of the changes they have executed or one of the changes has to be discarded. In the latter case, we can simply discard the pull request.

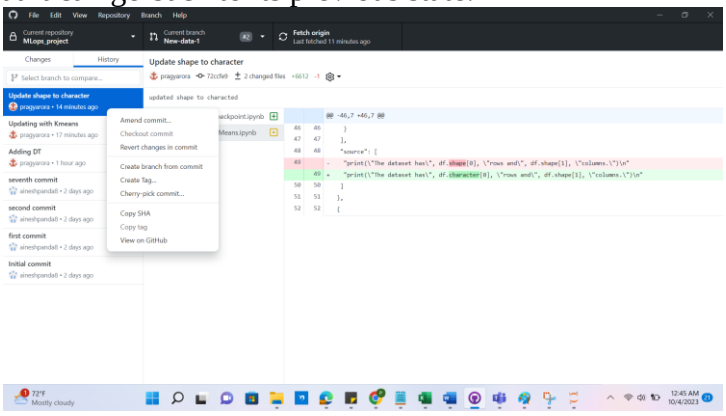
15. **History Tab**: This tab helps us analyze the changes that have been made in the past to understand our execution plan. If the comment while committing a change is written properly, one can benefit a lot from the History tab to understand the origin and resolution of the merge conflict.

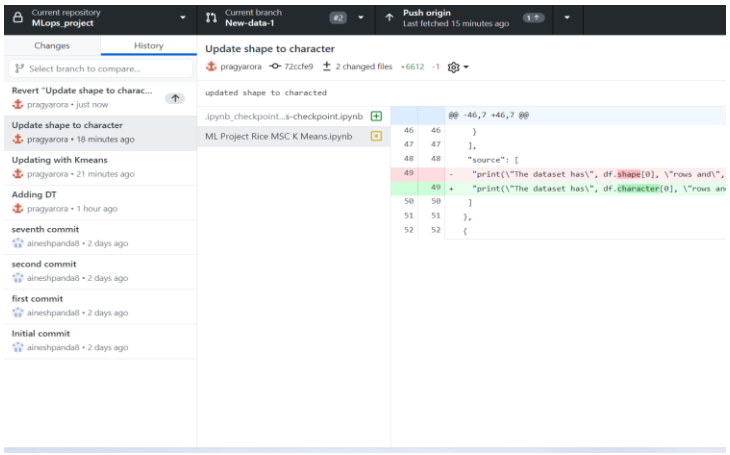


The yellow highlighted file shows that there are some issues in the commit and shows us the difference.



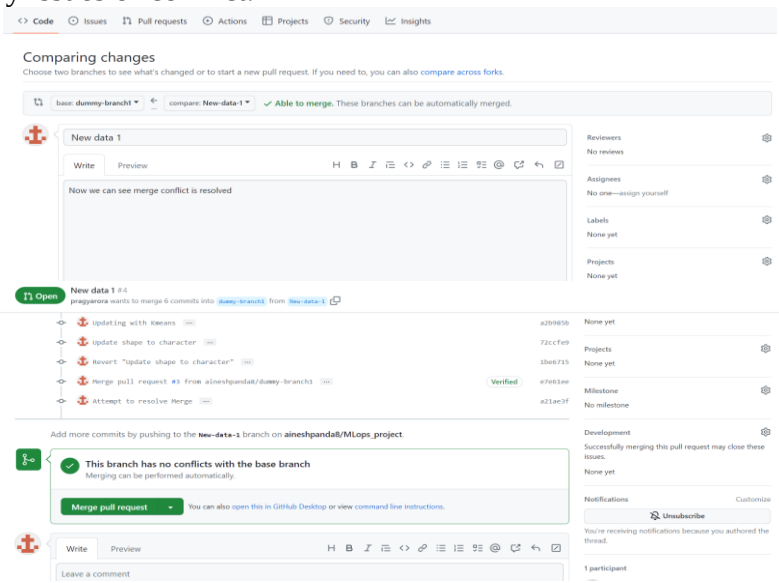
16. **Reverting Commit**: if we believe that the commit we have made is not right or want to first resolve the merge conflict before executing the change, we can go ahead and revert the commit so that it can go back to its previous state.



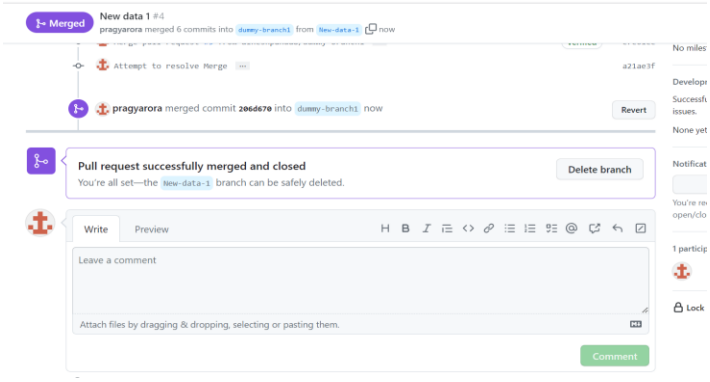


We publish this change now and check the status of the branch. We see that now it has reverted the change and just showing the KMeans file with the variable “shape” rather than “character”. We need to make sure to take a fresh full of the source branch into the New-Data-1 to keep up to date with the version of the Source Branch.

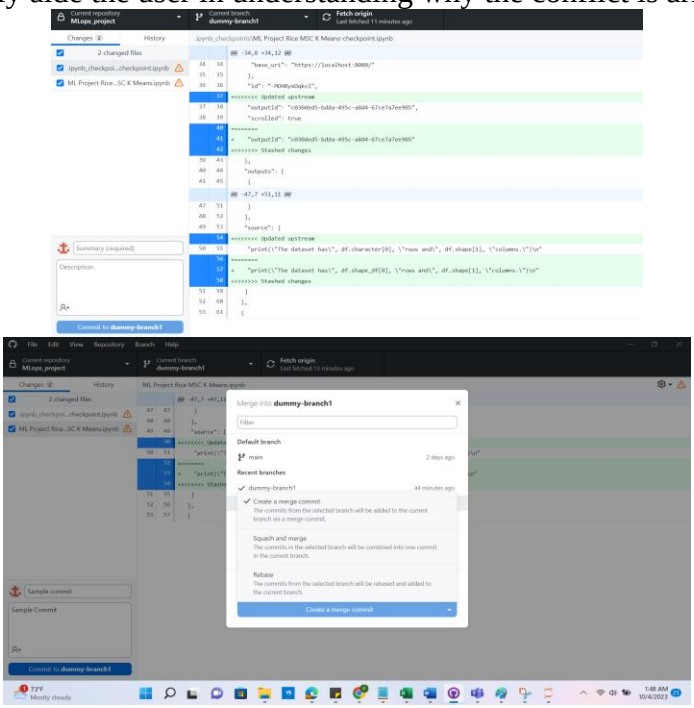
In a scenario where we wanted both the changes to be incorporated, we can take a fresh pull in the branch from the source branch and add changes. Now we can see that the merge conflict is resolved. A small step of taking a fresh pull before executing the steps can resolve so many issues of conflict.



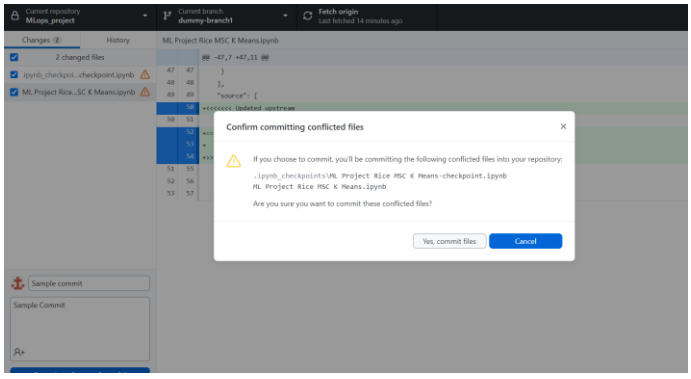
Now we see that the following merge is successfully executed and the new changes are visible.



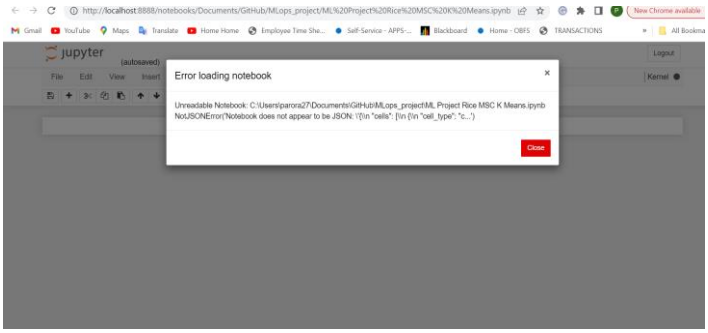
GitHub Desktop has it's own unique way to alert about a merge conflict. If a change is made to a file that will cause merge conflict, it will start giving warnings and on the forced switch will visually aide the user in understanding why the conflict is arising.



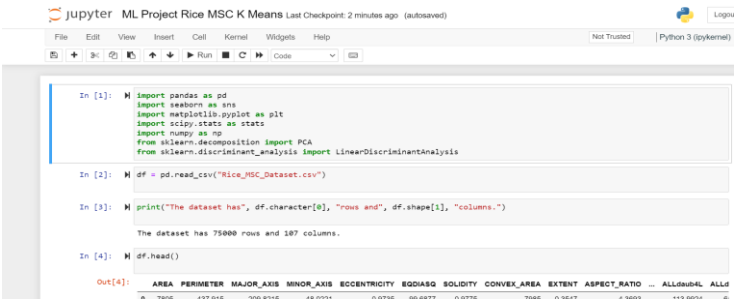
We are provided with multiple solutions. This makes it extremely beginner-friendly, who might have a hard time understanding the operations that need to be done to resolve conflicts.



A forced push ended up breaking our environment.



Either we can open the file and based on the arrows mentioned, add and subtract the parts we want in the final formal or revert the commit and make amendments. At this place, the steps are comparatively more simplified in the command prompt. But still, it is extremely user-friendly and requires a learning curve to understand its flow. Now our code file should reopen



Pros and Cons:

Pros:

1. *User-Friendly Interface*: GitHub Desktop provides an intuitive and user-friendly graphical interface, making it easier for both beginners and experienced developers to work with Git repositories.

2. *Cross-Platform*: It is available for both Windows and macOS, allowing developers to work on different platforms with ease.

3. *Seamless GitHub Integration*: GitHub Desktop is specifically designed for GitHub users, offering seamless integration with GitHub repositories and features.

4. *Visual Branch Management*: It offers a visual representation of branches, making it easier to create, switch between, and merge branches.

5. *Commit History Visualization*: You can easily view commit histories and changes made to files, helping you understand project development over time.

6. *Conflict Resolution*: It provides tools to help resolve merge conflicts, simplifying the process of merging changes from multiple contributors.

7. *Offline Access*: Unlike some web-based Git interfaces, GitHub Desktop allows you to work offline, which can be crucial in certain situations.

Cons:

1. *Limited Features*: GitHub Desktop may lack some advanced Git features and commands available in the command-line interface. Developers with complex Git workflows might need to switch to the command line for certain tasks.

2. *GitHub-Centric*: While it's great for GitHub users, it might not be as suitable for developers who prefer other Git hosting services or use Git independently of any platform.

3. *Resource Consumption*: Like any graphical application, GitHub Desktop consumes system resources, which can be a concern on older or less powerful computers.

4. *Learning Curve*: While it's user-friendly, there may still be a learning curve for those new to version control and Git concepts.

5. *Limited Customization*: GitHub Desktop offers less customization compared to using Git from the command line, which can be a drawback for power users.

6. There are a lot of places where a command prompt will be more useful, especially when the conflicts are complicated and require to use of commands such as “git-bisect”, “git-cherry-pic” or “git-reflog”. Even though GitHub desktop provides a very beginner-friendly conflict resolution understanding, for more complicated issues, the command prompt is more efficient.

GitHub Desktop Best Practices:

1. *Keep Repositories Small and Focused*: Create separate repositories for distinct projects or components to maintain clarity and organization.

2. *Use Descriptive Commit Messages*: Write clear and concise commit messages that explain the purpose of each commit. Follow a consistent format, such as starting with a verb in the present tense.

3. *Frequent Commits*: Commit changes frequently in small, logical chunks rather than making a single large commit. This makes it easier to track and review changes.

4. *Branch Naming Conventions*: Adhere to a consistent branch naming convention, such as "feature/feature-name" or "fix/issue-number," to easily identify the purpose of each branch.

5. *Pull Request (PR) Guidelines*: When creating a PR, provide a detailed description of the changes, the problem being solved, and any relevant context. Mention relevant issues or collaborators.

6. *Code Reviews*: Actively participate in code reviews for PRs, provide constructive feedback, and ensure code quality and consistency.

7. *Branch Cleanup*: Delete local and remote branches after they are merged and are no longer needed. This helps keep the repository clean.

8. *Use .gitignore*: Create a .gitignore file to specify which files and directories should be excluded from version control to avoid cluttering the repository with unnecessary files.

9. *Regular Pulls and Updates*: Regularly pull changes from the remote repository to keep your local repository up to date. This helps prevent merge conflicts.

10. *Commit Before Switching Branches*: Commit or stash your changes before switching branches to avoid losing work or creating merge conflicts.

11. *Merge Conflicts:* If you encounter a merge conflict, use tools like GitHub Desktop's conflict resolution tools or external merge tools to resolve conflicts efficiently.

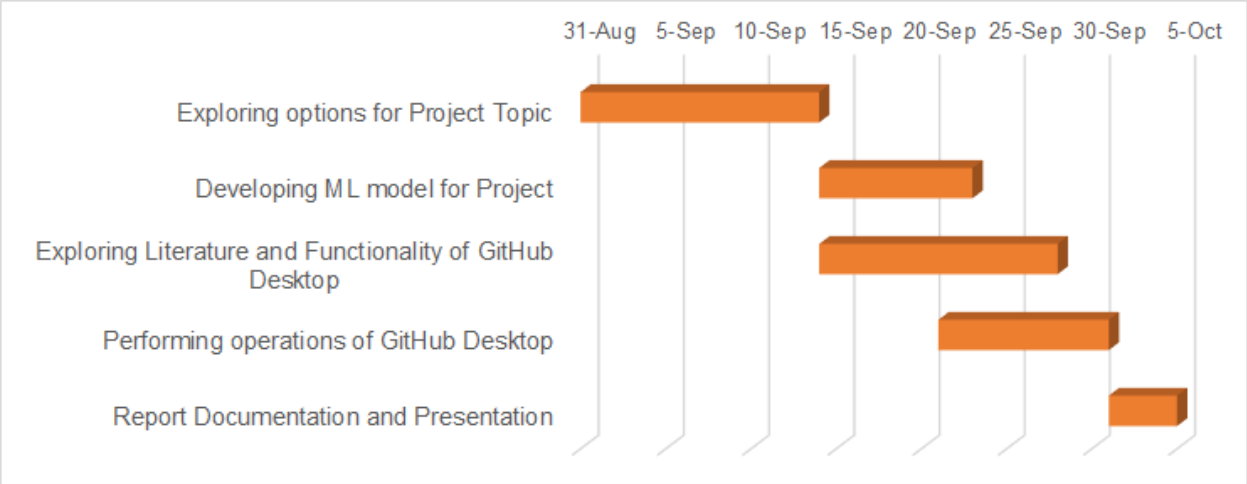
12. *Backup and Recovery*: Regularly back up your work by pushing changes to remote repositories. Familiarize yourself with Git recovery techniques like reflogs.
13. *Git Ignore Whitespace*: When reviewing changes in PRs, consider using the "Ignore Whitespace" option to focus on meaningful code changes rather than formatting differences.

14. *Monitor Repository Notifications*: Stay informed about activities in your repositories by enabling notifications. This helps you stay updated on issues, PRs, and discussions.

Lessons Learned:

1. While cherry-picking a commit, it can only be done to one commit at a time. It means that a commit from the branch can be replicated to another branch only and not to multiple branches on a single go.
2. While squashing commits together, if the commits have been cherry-picked to another branch, the squashing is not replicated in the other branch.
3. When publishing branches from GitHub desktop to GitHub server, it can also be done one at a time.
4. Merge Conflict is avoidable if regular pulls are taken from the source branch.

Project Plan:



References:

1. Brent Beer. (2018). Introduction to GitHub. Introducing GitHub. (n.d.). Google Books. pp:126-144. https://books.google.com/books?id=zpNFDwAAQBAJ
2. Daniel van Strien. (2016, June 17). An Introduction to Version Control Using GitHub Desktop. https://programminghistorian.org/en/lessons/retired/getting-started-with-github-desktop
3. GitHub Desktop Documentation, GitHub Docs. https://docs.github.com/en/desktop