Git

Introduction

Git is a free and open-source version control system. A version control system is software that helps software developers to work together and maintain a complete history of their work. It's a system that records changes to a file or set of files over time so that you can recall specific versions later.

As more code is added, the code stored in Git keeps changing. It's basically like a history tab for your code editor.
If you ever face some error and you don't know how to fix it, you can always revert back to a stable version of your code.

For example, if you develop an app, and you start rolling out various versions of your app. With each version, you want to incorporate extra features.
After rolling out two versions, suddenly, you discover a bug in the latest version. What will you do? The best option would be to roll back to the previous release, fix the 3rd version, and then roll it out again.

But for this, you'd need to change the code, which will disrupt the services for the current user base. In this type of situation, Git is very useful, as it keeps a track of all the changes in code, and makes reverting back very easy.

Understanding Terminologies

Repository: A repository is a collection of source code. It basically contains all of the project files and the entire revision history. You'll take an ordinary folder of files (such as a website's root folder), and tell Git to make it a repository. This creates a .git subfolder, which contains all of the Git metadata for tracking changes.

Commit and Staging: Git keeps a list of changes to files. So how do we tell Git to record our changes? Each recorded change to a file or set of files is called a **commit**.
Before we make a commit, we must tell Git what files we want to commit. This is called **staging** and uses the **add** command.

Now, you might ask why must we do this? Why can't we just commit the file directly? Let's say you're working on two files, but only one of them is ready to commit. You don't want to be forced to commit both files, just the one that's ready. That's where Git's add command comes in. We add files to a staging area, and then we commit the files that have been staged.

Branch and Merge: A branch is a version of the repository that **diverges** from the main working project. It is an essential feature available in most modern version control systems. A Git project can have more than one branch. This lets you more easily work with other developers, and gives you a lot of flexibility in your workflow. We can perform many operations on Git branch-like rename, list, delete, etc.
The git **merge** command facilitates you to take the data created by the git branch and integrate them into a single branch.

Master and Origin: Master is a naming convention for the Git branch. It's a default branch of Git. When you clone a project from a removed server, the resulting local repository contains a single branch, which is called, "**master**"
branch. It means that this branch is the repository's default branch. **origin** is an alias on your system for a particular remote repository. It's not actually a property of that repository. It's used instead of that remote repository's URL, which makes referencing much easier.

Checkout: The git **checkout** command is used to switch between branches in a repository.

Push and Pull: Pull is used to receive data from **GitHub**. It fetches and merges changes on the remote server to your working directory. The git pull command is used to make a Git pull.
The **push** command is used to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository. You should be cautious while pushing code, as pushing **overwrites** changes.

Fork and Clone: When you fork a repository, you create a copy of the original repository but the repository remains on your **GitHub account**. Whereas, when you clone a repository, the repository is copied onto your **local machine** with the help of Git.

GitHub: GitHub is a Git repository hosting service, but it also has many of its own features. While Git is a command-line tool, GitHub provides a Web-based GUI tool. It also provides access control and several collaboration features, such as wikis and basic task management tools for every project.

The flagship functionality of GitHub is "**forking**" – copying a repository from one user's account to another. This enables you to take a project that you don't have write access to and modify it under your own account.

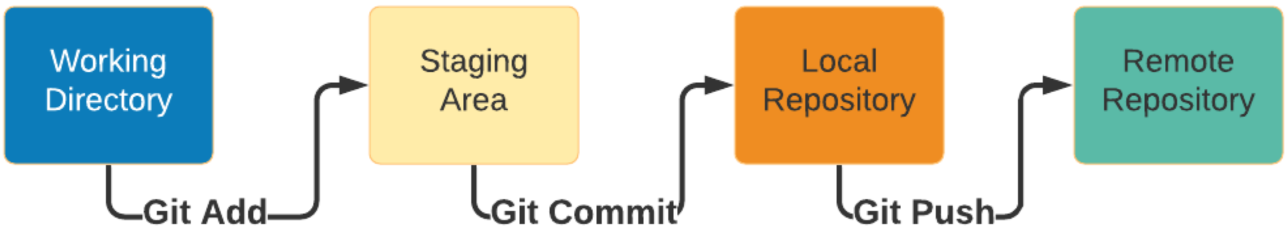
Git Workflow

If you consider a file in your Working Directory, it can be in three possible states:

- It can be **staged**. This means the files with the updated changes are marked to be committed to the local repository but not yet committed.
- It can be **modified**. This means the files with the updated changes are not yet stored in the local repository.
- It can be **committed**. This means that the changes you made to your file are safely stored in the local repository.

Now, let's understand the basic workflow of Git

1. You modify a file in the working directory.
2. You add this file to the staging area.
3. You perform a commit operation, which moves the file from the staging area.
4. Then you push, which permanently stores the changes to the Git repository.



Some basic commands:

- **git add** is a command used to add a file that is in the working directory to the staging area.
- **git commit** is a command used to add all files that are staged to the local repository.

- **git push** is a command used to add all committed files in the local repository to the remote repository. So in the remote repository, all files and changes will be visible to anyone with access to the remote repository.
- **git fetch** is a command used to get files from the remote repository to the local repository but not into the working directory.
- **git merge** is a command used to get the files from the local repository into the working directory.
- **git pull** is the command used to get files from the remote repository directly into the working directory. It is equivalent to a git fetch and a git merge.

Installing Git

Before using Git, make sure you have Git installed in your system. First, create a GitHub account and then install Git on your system.
Installing Git on Mac and Linux-based systems is very easy.
If you are on a **Mac**, fire up the terminal and enter the following command:

```
$ git --version
```

This will prompt open an installer if you don't already have Git. So, set it up using the installer. If you have git already, it'll just show you which version of git you have installed.
If you are running **Linux**, enter the following in the terminal:

```
$ sudo apt install git-all
```

If you are on **Windows**:
Visit https://gitforwindows.org/ and install Git from here.

Using Git

Now, we'll show how to use Git, using an example. Create a GitHub repository.
When you'll first start Git, it'll ask for your username and email, with which you created your GitHub account. You'll need to mention your Git username and email address since every Git commit will use this information to identify you as the author.
If Git doesn't't' remind you to enter the details, you can use the commands below to set your username and email:

```
$ git config --global user.name "YOUR_USERNAME"
$ git config --global user.email "YOUR_EMAIL”
```

You can verify the above information using this command:
```
$ git config --global  --list
```

First, you'll need to navigate to any folder, then you'll have to initialize Git to place it under it.
```
cd /Home/Folder
```

The **git init** command is used to create a new blank repository. It is used to make an existing project a Git project. To create a blank repository, open the command line on your desired directory and run the init command as follows:
```
git init
```

After this, it'll say:
```
Initialized empty Git repository in C:/Home/Folder/.git/
```

Now we can create and add files on this repository for version control. We've created a text file inside this folder to show the working of Git. You can use any file instead.

To add files to the repository, run the git add command as follows:
```
git add .
```
This adds all the files in the local repository and stages them for commit.
If you want to add a specific file:
```
git add abc.txt
```

To see which files are to be committed, you can run the **git status** command:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   abc.txt.txt
```

To commit changes to add files to the local repository run **git commit** command:
```
$ git commit -m 'First Commit'
[master (root-commit) 6471664] First Commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 abc.txt.txt
```

The message in the ' ' is given so that the other users can read the message and see what changes you made. Suppose if you made some error in your code, and you want to uncommit the changes, you can run this command to remove the most recent commit:

```
git reset HEAD~1
```

When you make some changes in your files and save them, they won't be automatically updated on GitHub. All the changes made are saved to the local repository. To update the changes to the remote origin/master:

```

git remote add origin remote_repository_URL#
```

where remote URL is Git's fancy way of saying "the place where your code is stored."

That URL could be your repository on GitHub, or another user's fork, or even on a completely different server

You can only push to two types of URL addresses:

- An HTTPS URL like https://github.com/user/repo.git
- An SSH URL, like git@github.com:user/repo.git

Git associates a remote URL with a name, and your default remote is usually called the origin. You can use the command **git remote set-url** to change a remote's URL. To push the changes to the remote repository run:

```
$ git push -u origin master
```

Now the **git push** command pushes the changes in your local repository up to the remote repository you specified as the origin. Now, if you and check your repository on GitHub, it'll show all your files along with all the commits made by you.

And that's how you can add the files to the repository, which you'll create on GitHub.

When you'll start making changes in your files and save them, they'll not match the last version that was committed to Git. You can see those changes by:

```
$ git diff
```

You can revert back to the previous version by entering:

```
$ git checkout .
```

Or for a specific file

```
$ git checkout -- <filename>
```

If you want to see all the commits made, you can use the **git log.**

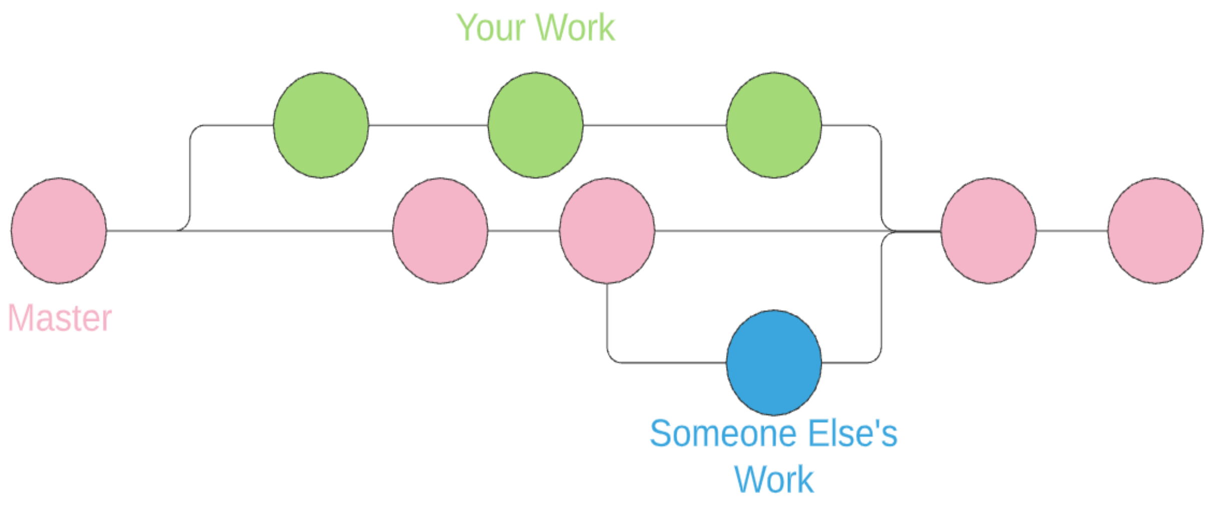
```
$ git log
commit 64716647e27b59ace096c4331b0497989662a485 (HEAD > master)
Author: abc <abc@gmail.com>
Date:   Sun June 7 01:33:31 2020 +0530

     First Commit
```

If you want to avoid committing any folders, you can tell git to ignore them by adding a **.gitignore** file.

Git Branching

Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to contain the changes. This makes it harder for unstable code to get merged into the main code.



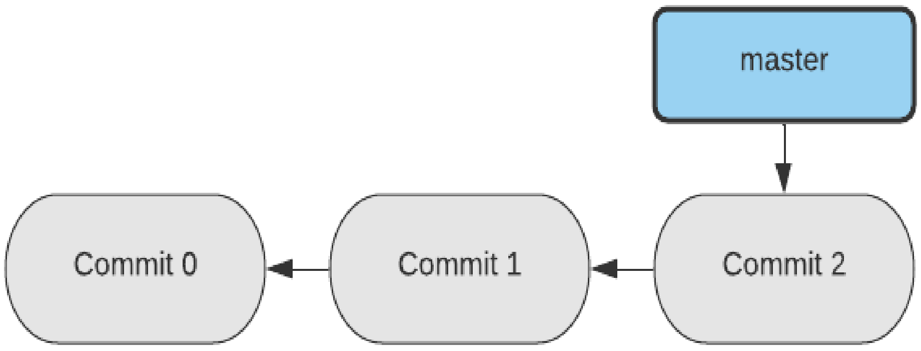
The diagram above visualizes a repository with two isolated lines of development, one for your code, and someone else's code.

By developing them in branches, it's not only possible to work on both of them in parallel, but it also keeps the main master branch free from any errors. We can also merge branches when we need to incorporate the changes in the master branch.

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand-new working directory, staging area, and project history.

The git branch command lets you create, list, rename and delete branches. The git branch command is tightly integrated with the git checkout and git merge commands.

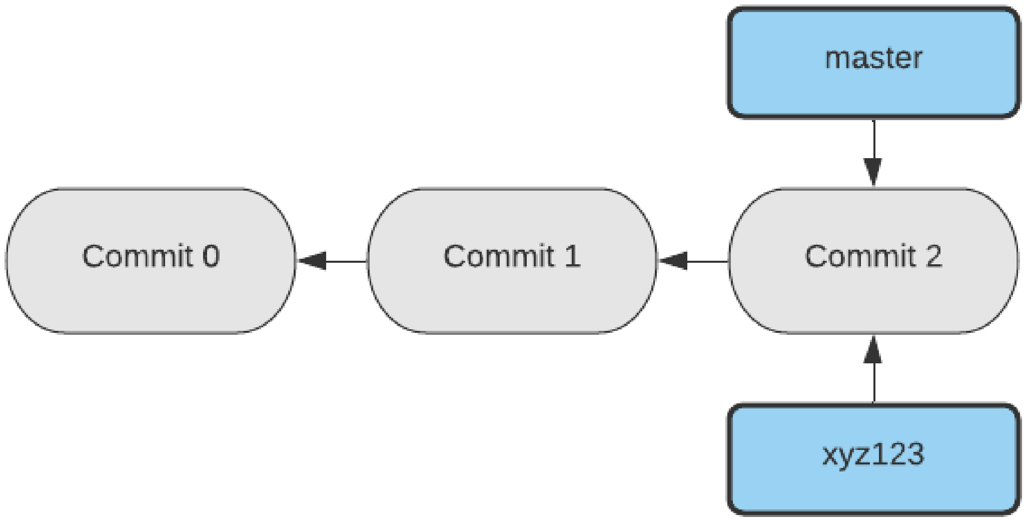
Let's say you're working on a website and you already have some commits already on the master branch.



You're going to work on an issue, for which you'll need to create a new branch named xyz123 and switch to it at the same time.

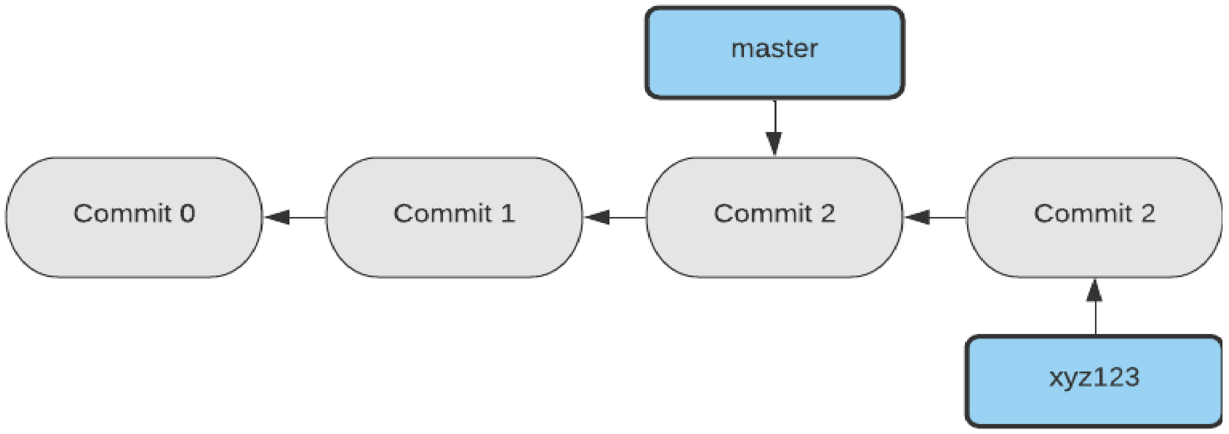
You can run the **git checkout** command with **-b:**

```
git checkout - b xyz123
Switched to a new branch "xyz123"
```



The HEAD points out the last commit in the current checkout branch.
Let's say, now you do some work and commit your changes.

```
vim index.html
git commit - a - m 'Create new navbar'
```



Doing so moves the xyz123 branch forward because you have it checked out (that is, your HEAD is pointing to it):

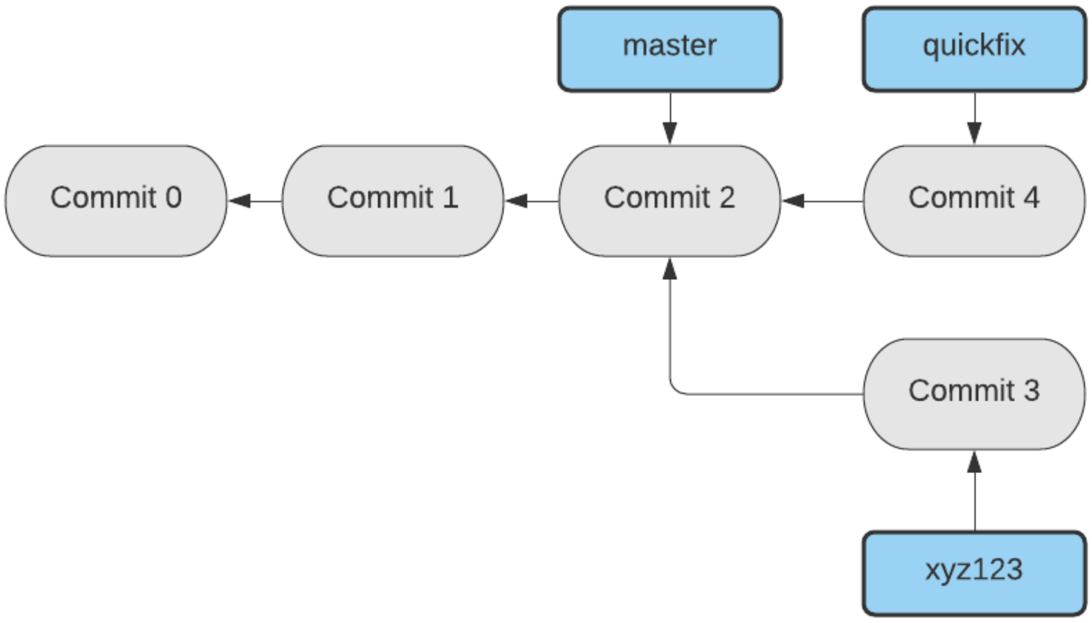
Now let's assume that some error appears on your website, and you need to fix it immediately. With Git, you don't have to put on lots of efforts to revert those changes before you can work on applying your fix. All you have to do is switch back to your **master** branch.

```
$ git checkout master
Switched to branch 'master'
```

At this point, your project working directory will exactly the way it was before you started working on that error. Git resets your working directory to look like it did the last time you committed on that branch.

Now, you need to create a **quickfix** branch on which you'll work until the fix is completed:

```
$ git checkout -b quickfix
Switched to a new branch 'quickfix'
$ vim index.html
$ git commit -a -m 'Fix broken navbar'
[hotfix 1fb7853] Fix broken navbar
 1 file changed, 2 insertions(+)
```



After running some tests and making sure that the quickfix is what you want, you'll finally merge the quickfix branch back into the master branch to deploy to production. We'll do this with the **git merge** command:

```
$ git checkout master
$ git merge quickfix
Updating f42c576..3a0874c
```

```
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

We'll delete quickfix branch as we don't need it now.
We can delete it using the following command:

```
$ git branch -d quickfix
Deleted branch quickfix (7a0365c).
```

Now that our fix is deployed, we can switch back to xyz123 to do the work we were doing before.

```
$ git checkout xyz123
Switched to branch "xyz123"
$ vim index.html
$ git commit -a -m 'Finish the new navbar'
[xyz123 b1638ba] Finish the new navbar
1 file changed, 1 insertion(+)
```

It's worth noting here that the work you did in your quickfix branch is not contained in the files in your xyz123 branch.
Now, you've completed the work and your branch xyz123 is ready to be merged with master branch. For that you have to check out the branch you wish to merge into and then run the **git merge** command:

```
$ git checkout master
Switched to branch 'master'
$ git merge xyz123
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Now that your work is merged in master, you don't need xyz123, you can get rid of it.

```
$ git branch -d xyz123
```

Merge Conflicts

Sometimes, the merging process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them. Then a merge conflict will arise, which looks something like this:

```
$ git merge xyz123
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

Git will pause the merging process until you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status.

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git will automatically add conflict resolution markers to the files that have conflicts, which will help us identify the conflict and resolve it manually. Assume that your file has a section which looks something like this:

```
<<<<<<< HEAD:index.html
<nav>
    <ul id="list">
        <li>1</li>
        <li>2</li>
        <li>3</li>
    </ul>
</nav>
=======
<nav id="navbar">
    <ul id="list">
        <li>1</li>
        <li>2</li>
        <li>3</li>
        <li>4</li>
    </ul>
</nav>
>>>>>>> xyz123:index.html
```

This means the version in HEAD (your master branch) is the top part of that block (everything above =======), while the bottom part is the version in xyz123 branch. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the above code by:

```
<nav id="navbar">
    <ul id="list">
        <li>1</li>
        <li>2</li>
        <li>3</li>
        <li>4</li>
    </ul>
</nav>
```

As you can see, the resolution has some part of both the versions, and the <<<<<<<, =======, and >>>>>>> lines are completely removed!

As we have resolved the conflict, we can mark it resolved by running the **git add** command. Staging a file marks it resolved in Git. You can run git status again to verify that all conflicts have been resolved. If you're happy with that, and you verify that everything that had conflicts has been staged, you can type **git commit** to finalize the merge commit.

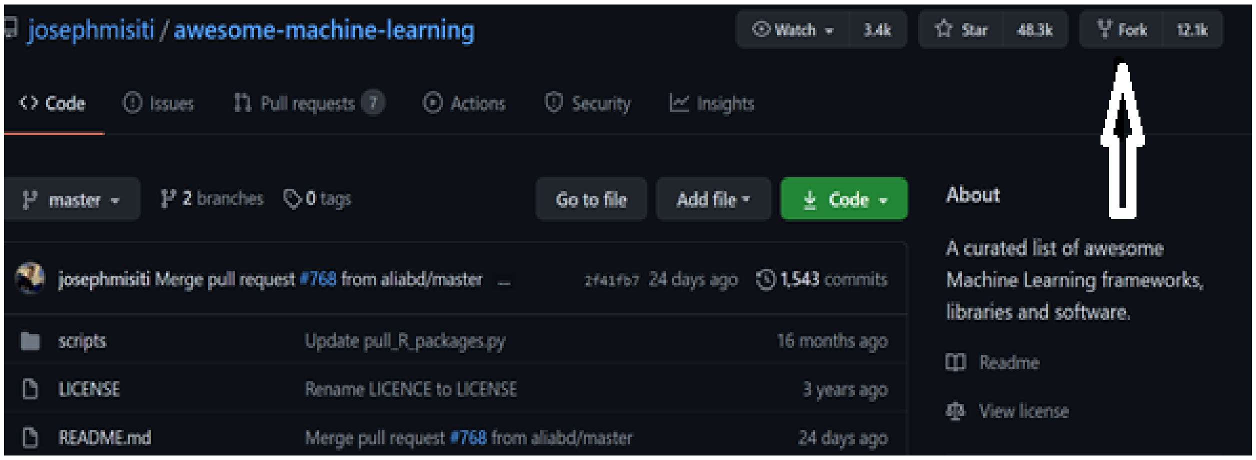
Git Fork and Pull Requests

A fork is a copy of a repository. Forking a repository will allow you to freely test and debug without affecting the original project.
Forking is not a Git feature; it's a feature of GitHub.

Following are the reasons for forking the repository:

- Propose changes to someone else's project.
- Use an existing project as a starting point.

The forking and branching are excellent ways to contribute to an open-source project. These two features of Git allow the enhanced collaboration on the projects. Forking is a safe way to contribute. It allows us to make a rough copy of the project. We can freely experiment on the project. After the final version of the project, we can create a pull request for merging.



josephmisiti Merge pull request #768 from aliabd/master … 2f41fb7 24 days ago 1,543 commits

scripts	Update pull_R_packages.py	16 months ago
LICENSE	Rename LICENCE to LICENSE	3 years ago
README.md	Merge pull request #768 from aliabd/master	24 days ago

A curated list of awesome Machine Learning frameworks, libraries and software.

Readme

View license

We can only fork other repositories, as only shared repositories can be forked. If someone wants to fork a repository, then he must log in with his account.
We can see the fork option at the top right corner of the page. By clicking on that, the forking will start. It will take some time to make a copy of the project for other users. After forking is completed, a copy of the repository will be copied to your GitHub account. It will not affect the original repository. You can freely make changes and then create a pull request for the main project.

Sometimes people think that the fork is same as the clone command because of their property. Both commands are used to create another copy of a repository. But a significant difference between the two is that the fork is used to create a server-side copy, and clone is used to create a local copy of the repository.

Pull requests are a mechanism for a developer to notify team members that they have completed a feature. If you're collaborating on a work assignment with your colleagues or contributing to an open-source project, there are chances that you'll be operating in the following scenario. You make local code changes and then submit those changes to a remote project maintainer/reviewer for review before those changes are implemented, or merged.
This is called a pull request; you are requesting that someone reviews and approves your changes before they become final.

You can read more about pull requests from the link below:

https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests