Using Git to Track Changes Using Local Repos

Bruce D. Marron
Project No. 2018NWDS006
©NW Data Science, LLC, Portland, Oregon 97214

August 19, 2018

About This Document

This document, "Using Git to Track Changes Using Local Repos" provides a version control system (VCS) for the propriety datasets, scripts, and other important documents of FilmProfit, LLC. The goal of the VCS is to maintain a highly accurate, easily managed, and readily accessible record of ALL changes to ALL important document files. Important document files would include proprietary datasets and scripts held by FilmProfit, LLC.

General Concepts

The VCS uses Git, a free and open source distributed VCS designed to handle everything from small to very large projects with speed and efficiency. Version control has become highly codified with the rise of software configuration management (SCM): the task of tracking and controlling changes in software. SCM is part of the larger cross-disciplinary field of configuration management and ultimately, of QAQC.

Usually there is a remote (offsite) storage repository and a local (onsite) working repository (in your computer). The idea is that you would work locally, make changes, then push the changes upstream to the remote repository. Our VCS has both the remote repository and the local repository on the same machine. A .git folder will be installed into a directory where the files that need to be tracked for changes are located. Another .git folder will be created to act as an remote repository on the local machine. Git, of course, has the ability to maintain offsite remote repositories as well. Typically offsite repositories are stored in GitHub. Note that Git and GitHub are not the same.

After working on a document (i.e., changes are made), a series of git commands will ensure that every single change in every single document is tracked. Git tracks changes with SHA-1 (Secure Hash Algorithm) tags. That is, each time a single file is changed or a set of files is changed and then committed to Git for version control, Git tags the changes with a SHA-1 tag. As explained by Google, SHA-1 is a cryptographic hash function which takes an input and produces a 160-bit (20-byte) hash value known as a message digest. The message digest is typically rendered as a hexadecimal number, 40 digits long.

The General Process

Git processes file changes in a sequence of actions. If you have created a new file and then added that file to a git-monitored repository, the file and all of the subsequent changes to that file can be tracked with Git by the following sequence of Git-specific actions:

track ==> stage ==> commit ==> push

For a file that is already in a git-monitored repository, the file and all of the subsequent changes to that file can be tracked with Git by the following sequence of Git-specific actions:

stage ==> commit ==> push

When you make a new commit followed by a push to the remote repo, Git stores a snapshot of your repository at that specific moment in time; later, you can use Git to go back to an earlier version (snapshot) of your project. Note that the GUI-based, "gitk" nicely provides a visual look at the details and the structure of the remote repo.

Editing of any ASCII file (.txt, .Rmd, .csv, .tex) can be done in Notepad++. Notepad++ allows for easy changes between the three kinds of end of line characters:

```
Carriage Return (MAC pre-OSX) == (CR | \r | ASCII code 13)
Line Feed (Linux, MAC OSX) == (LF | \n | ASCII code 10)
Carriage Return and Line Feed (Windows) == (CRLF | \r\n | ASCII code 13 and then ASCII code 10}
```

Notepad++ allows for easy changes between the kinds of encodings: ANSI | UTF-8 | UTF-8 BOM | UTF-8 BOM | USC-2 BE BOM | USC-2 LE BOM. Lastly, Notepad++ has a large number of script language settings.

The following sections provide the details of the VCS: setting Git global variables, setting up a remote repo, setting up a local repo, and tracking changes with Git.

The Installation

Note that commands (in typewriter font) are executed in the Git BASH shell. Simply open GitBash by clicking on the GitBash icon in the taskbar. You will see a new window with a command line prompt. Enter the commands at this prompt.

Setting Git globals

Make sure some fundamental global variables are set.

```
git config --global user.name <your GitHub username> &&
git config --global user.email <your GitHub registered email> &&
git config --global core.editor notepad &&
git config --global http.sslVerify false
git config --list
(type 'q' to exit)
```

Set-up a remote repository

A remote repo for our VCS will be a special directory (folder) that contains only a "bare" Git set of files. A unique remote repo is needed for each working repo. A working repo is simply a directory (folder) that contains Git-tracked files. Note that the name of the remote repo must end in ".git".

```
mkdir -p ~/Desktop/GitRemoteRepos/Local1.git &&
cd ~/Desktop/GitRemoteRepos/Local1.git &&
git init --bare
```

Set-up a local repository

A local repo is one that contains Git-tracked files. This is the repo in which files are added, worked on, and ultimately modified. It is assumed that the files to be Git-tracked already exist in some directory (folder). To track files, Git will install a '.git' folder.

```
cd <path to datasets folder>
git init
ls -a
git status
git add --all
git status
git commit
(suggest day-month-CommitNumber as message; eg 11Aug-1)
git remote add origin ~/Desktop/GitRemoteRepos/Local1.git
(if make a mistake use: git remote set-url origin ~/Desktop/GitRemoteRepos/Local1.git )
git remote -v
git push -u origin master
```

Basic workflow

After making changes to one or more files, Git-tracking is activated by staging the files, commiting the files to the local repo, and then pushing the changes to the remote repo.

```
git status
git add --all
git status
git diff --cached
(type 'q' to exit)
git commit
(can also use this to do stage AND commit simultaneously: git commit -a)
git push origin master
gitk
(may need to open/close gitk a few times)
git log
git log -1
git log -2
```

Just in case ...

One of the most useful features of Git is its ability to “undo” mistakes. However, “undo” can mean many slightly different things in Git. That said, it is possible to undo virtually anything in Git. The following are some typical mistakes and their remedies.

Bad changes inadvertently saved to a file

This will undo everything in the corrupted file back to the way it looked in the last commit. Ideally, the basic workflow becomes automatic so changes are committed frequently.

```
git checkout -- <bad filename>
(a "Do you want to reload file" message is generated)
```

Bad changes in commit but BEFORE the commit is pushed to the remote repository

This will undo the commit but keep your changes for editing. You can then re-commit.

```
git reset HEAD~1
```

Bad changes in a commit JUST pushed to the remote repo

There's a problem in at least one of the files just commited. Before reverting, any working changes must be either committed or stashed. The a new, "reverse" commit will be added taking the repo back to just before the bad push. You can fix things and then re-commit and re-push.

```
git stash
git log -1
git revert <the most recent SHA>
git stash pop
(a "Do you want to reload file" message is generated)
```

Bad changes in commits PREVIOUSLY pushed to the remote repo

First, look at former states of the repo in either 'gitk' or by checking out specific SHA tags (i.e., the state of the repo after a set of commits). Return to most recent state of the repo (master). Reset the repo to a former state (ie any prior commit) using its SHA tag.

```
gitk
git log
git checkout <some SHA>
git checkout master
git reset --hard <SHA tag that corresponds to the repo in the desired state>
```

Complete disaster

The entire folder containing the precious data is corrupted or destroyed.

```
mkdir ~/Desktop/<name of rescue folder>
cd ~/Desktop/<name of rescue folder>
git clone ~/Desktop/GitRemoteRepos/Local1.git
```