Using version control to facilitate a reproducible and collaborative workflow in acoustic phonetics

Nay San
Macquarie University
nay.san@mq.edu.au

Abstract

This paper outlines the benefits of using a version control system (VCS) in an acoustic phonetics workflow. Selected features of Git, a VCS, are introduced and explained in terms of their relevance to facilitating communication and reproducible research, especially in projects with multiple contributors. A workflow, as used in two current projects on Australian Aboriginal languages, is provided as an illustrative example. While the paper focusses on the Git VCS with Praat as the annotation tool, the workflow may be easily adopted using other VCS tools (e.g. Mercurial) and annotation tools (e.g. ELAN, EMU).

Index Terms: collaborative annotation, reproducible research, corpus management

1. Introduction

1.1. Role of annotations in speech corpora

A large part of any acoustic phonetic project lies in deriving linguistically relevant information from recorded audio or video signals. The information derived, for example, may be orthographic or phonetic transcriptions, temporal locations of segment boundaries, or pitch/formant tracks.

Ultimately, the analysis of such annotations generally constitute the results of interest, and various platforms have been proposed to facilitate reproducible analyses. For example, Alveo [1] provides the Galaxy workflow engine, through which analyses may be published as, essentially, an interactive flowchart. This allows for each analysis step to be reproduced on demand.

While producing documented and reproducible analysis workflows are highly important, the same aspects in annotation workflows have received little attention—despite the fact that analyses depend heavily on these annotations.

Annotation workflows can often be equally complex multi-step and multi-level processes. Consider a single aspect, automation. Creating annotations may involve some degree of automation (fully-, semi-automated, or fully-manual). For example, transcriptions may involve using only speech-to-text software (fully-automated), its use with supervision from a human transcriber (semi-manual), or rely solely on human transcription (fully-manual). Additionally, a given project may also choose to employ a mix of methods to create the desired annotations. For example, word-level transcriptions may be first manually transcribed, while segment-level boundaries are then derived via forced-alignment.

Given that annotation processes can be complex, it is very likely that a given set of annotations contains a number of errors. Indeed, annotation errors are well documented even in large, publicly-released corpora [2]. Thus, annotation data cannot be expected to remain entirely static, as errors should be corrected as they are discovered.

However, for analyses to be reproducible, the annotations used in the analyses must not change. Thus, it appears that annotation data are required to be both static and dynamic, at the same time. The use of version control, however, can satisfy this seemingly paradoxical requirement.

1.2. Collaborative workflows

Projects may also involve several annotators in differing locations. One way to handle this complexity has been to establish a central annotation server [3], where users create and modify annotations through a web application. This solution, however, may not be appropriate where contributors do not have frequent and/or fast access to the server (e.g. those in the field).

The common solution for collaboration appears to be to use a cloud hosting service, such as Dropbox or Google Drive, to synchronize annotation files (`.TextGrid`, `.ELAN`, etc.). While these services do offer some level of version control (revert to old versions), multiple contributors editing a single file often results in conflicting copies of the files (e.g. `fileA.TextGrid`, `fileA (1).TextGrid`). Not only do most cloud hosting applications lack explicit notification of conflicts as they emerge, they do not provide a convenient method for resolving them—especially when the conflicting files have to be merged together.

Thus, collaboration in annotation workflows not only require a way to share annotation files, but also to merge the resulting annotations from the various collaborators.

1.3. Overview

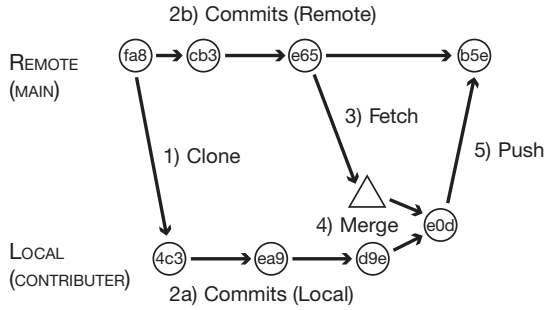
The annotation of speech data from audiovisual signals can be a complex, cyclical and distributed process. Facilitating this process thus requires a system for tracking, documenting and synchronising changes. Additionally, the system should also allow for changes to be made locally using standard annotation software (e.g. Praat [4], EMU [5]).

Through a description of Git, a version control system (VCS), this paper will show that a VCS is exactly what is required to satisfy the requirements stated above. Section 2 provides an introduction to Git, first defining some key terminology in Git, then describing the relevance of some features of Git in the acoustic phonetics workflow (Subsections 2.1-2.4). Section 3 provides an illustrative example of two acoustic phonetics projects using Git in the their workflows. Finally, the benefits of using a Git VCS in the workflow are summarised, and potential issues are highlighted in the discussion (Section 4).

2. Git

Git is a VCS that was developed in 2005 by Linus Torvalds, the creator of the Linux kernel. As the Linux kernel is free and open source, its development is driven by contributions from a large number of individuals. This requires a system for allowing the development to be distributed among many contributors, a way to review these contributions, and easily merge various contributions together into the main project.

A key feature of the Git VCS is that the system may take various forms, and it is not constrained to one 'standard' workflow. Various projects may adopt one that suits them best, and integrate Git accordingly. Figure 1 provides an illustration of some steps involved in adding a contribution to a relatively simple project model.



Note. Each circle represents a set of registered changes, a 'commit', which are given a unique identifier (text inside circle, abbreviated to 3 characters)

Figure 1: *A simplified illustration of the Git contribution process*

Clone. In Step 1), the contributor *clones* the project from an external source onto their local computer (analogous to downloading the files).

Commit. In Steps 2a) and 2b), various changes are registered or *committed* to the project. Commits can be thought of as the registration of significant changes. A project can be reverted back to a specific commit at any point in the future (i.e. undo various changes). Notice that Steps 2a) and b) can be done in parallel, and creation of commits require no communication between the local and remote sources.

Fetch. In Step 3), the contributor *fetches* changes from the remote source. This allows the contributor to view all the changes that have taken place since the last synchronisation.

Merge. In Step 4), the contributor *merges* their local changes with those fetched from the remote repository. Conflicting changes between the local and remote sources may prevent a successful merge. The contributor must resolve these conflicts during the merging process.

Push. In Step 5), following a successful merge, the contributor's changes can be integrated into the main project by *pushing* these changes up to the remote source (analogous uploading the files).

2.1. Commits: registered sets of changes

As mentioned, commits consist of registering various changes to files with the Git VCS. While 'Track Changes' in a Microsoft Word document, for example, might show *what* changes were done (and by whom and when), the context for the change, or *why*, is not explicitly required. Though, the why is very often a key piece of information. Moreover, Git allows the logical grouping of changes according to such information.

Consider an illustrated commit below, shown in Figure 2. The difference, or 'diff', between consecutive versions of two TextGrid files indicate *what* changes have taken place. For instance, the annotator John Doe has edited an annotation from 'balap' to 'palap' in File1.TextGrid (using Praat), and this edit has resulted in a change on Line 20 of the file.

```
Commit:  ea9
Author:  John Doe
Date:    9 June 2016, 2:05 pm
Message: Stop voicing not contrastive. Updated
         transcription procedure uses voiceless
         allophones.

Diff from 4c3 to ea9:

File1.TextGrid
19 19   ...
20    - text = "balap"
   20 + text = "palap"
21 21   ...

File5.TextGrid
33 33   ...
34    - text = "golot"
   34 + text = "kolot"
35 35   ...
```

Figure 2: *An illustration of a commit (ID: ea9) and the diff 'difference' in two TextGrid files between commits 4c3 and ea9*

A similar edit has also seen a change on Line 34 of File5.TextGrid. Changes to the two files have been registered under a single commit, ea9—with the reason for the changes provided in the required commit message. Commits can thus entail not only single changes, but sets of changes across multiple files. Conveniently, Git allows both single files (e.g. only File1.TextGrid) or entire sets of changes (e.g. both files) to be easily reverted to previous versions.

Commits thus provide the primary means of tracking and documenting significant changes to the various files in a given project. The flexibility in reverting allows for commits to be either entirely, or only partially, undone in the future.

2.2. Tags: noteworthy commits

A specific commit may be 'tagged' with some additional metadata. In software development, this is generally used to flag released versions of software, e.g. v 1.1.5. Analogous to software releases, tags can be used to note the exact versions of annotation data used various research output (e.g. ICPHS2015, SST2016).

The tagged version can be 'checked out' easily by anyone wishing to access an older state of the data. Essentially, a given analysis can continue to access the state of the data at a specific timepoint even without reverting current project files to that timepoint.

2.3. Distributed sources, controlled synchronisation

Cloud filesharing services such as Dropbox or Google Drive—unless paused—propagate changes across all synchronised folders immediately when the changes are performed. While

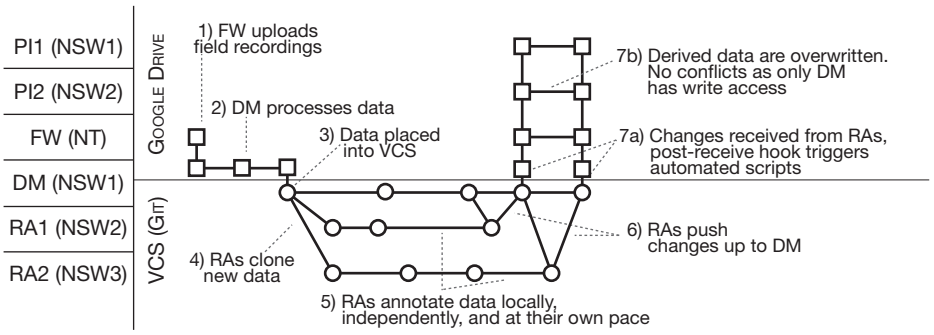


Figure 3: *Annotated illustration of a workflow in the Alyawarre/Anmatyerre Dictionary Recordings (AADR) project. Data are shared by various parties: investigators (PI1, PI2), fieldwork sources (FW), data manager (DM), and research assistants (RA1, RA2) across multiple locations (NT, NSW1, 2, 3)*

in most cases, this may be desired, a level of control over which changes and when they are synchronised may be desired in others. Moreover, immediate synchronisation of changes may create several conflicting copies if two annotators need to work on the same file concurrently.

With the Fetch-Merge-Push procedure of Git, annotators may retrieve (fetch) and send (push) changes in a controlled manner. In the majority of cases, Git is able to automatically merge changes to one file from multiple sources. Should a merge fail, Git alerts the user of the exact file(s) and line(s) within the file(s) preventing the merge; making conflict resolution relatively straightforward.

Thus, the use of version control reduces the overall clutter in project directories, eliminating not only the need for keeping manually-versioned files (e.g. 20160608-data-final-1.csv), but also conflicting copies from arising in shared directories.

It should also be noted that the use of Git does not prevent the use of services such as Dropbox or Google Drive. A portion of the project files may lie in a version controlled folder, while others lie in a cloud sync folder, e.g. in Dropbox. The illustrative example below will demonstrate such a use case.

2.4. Hooks: triggers for automation

As certain Git actions are associated with notable changes in the state of various files, Git hooks provide a mechanism of triggering follow-up actions when certain Git actions occur. For instance, a post-commit hook is an executable script that is run by the Git VCS after a user has created a new commit. As Git is language-agnostic, these scripts may be written in any language (e.g. Python, R).

While such hooks may be used for a variety of purposes (e.g. e-mail alerts, data validation), they provide a means of monitoring the state of a set of annotation data. For example, a simple script generating proportions of .wav files with and without corresponding .TextGrid files can provide a rough estimate of completed annotations. As the script is triggered after every commit, these proportions update automatically.

Additionally, monitoring annotation data may involve some simple extraction or data transformation procedures, e.g. from nested to tabular data. The benefits of generating such data early in the annotation process are two-fold. Firstly, since each commit generally consists of small sets of changes, annotation errors produced in the immediate or recent set of commits can be detected and sourced easily, as well as fixed promptly. Secondly, the extraction and transformation scripts can be repurposed for the data wrangling scripts in various analyses.

In other words, leveraging automated scripts to monitor annotation data can significantly reduce the considerable amount of time that is often spent validating and transforming data into appropriate forms prior to most analyses.

3. Illustrative example

Kaytetye, Alyawarre and Anmatyerre are three languages from the Arandic family, spoken in the region surrounding Alice Springs, NT. A core part of the Kaytetye Phonology (KPHON) and Alyawarre/Anmatyerre Dictionary Recordings (AADR) projects involve establishing data repositories of (mainly) audio recordings around the three languages. The data from these repositories are to be used in generating language resources (e.g. multimedia dictionaries), and to establish quantitative accounts of various phonological structures and processes.

The dictionary recordings in AADR consist of various speakers reading aloud the dictionary headwords in citation form, with several repetitions per headword. One goal across the two projects is the phonetic transcription of all the dictionary words by at least two transcribers. The transcription process is blind, in that annotators are not given the identity of the words, orthographic or otherwise.

Figure 3 illustrates an example of an annotation workflow in the AADR project; from receiving fieldwork recordings to the generation of the first set of derived data. In Step 1), fieldwork recordings are uploaded to a shared Google Drive. In Step 2), the data manager (DM) splits the lengthy recording sessions (e.g. 20160503-Wa-We.wav) into individual words (e.g. wampe.wav) for archiving. In Step 3), the single word files are anonymised (e.g. wampe.wav to file07.wav). These files are then received by the research assistants (RAs) for transcription (Step 4). As these files are cloned by the RAs through Git, all changes made by the RAs are tracked through Git commits (Step 5). When a portion (or all) files have been annotated, RAs may send the changed files back to the DM (Step 6).

When changes are received by the DM, a post-receive hook runs a number of scripts (Step 7), producing various reports. Table 1 displays the first 4-lines of data from such a automatically generated report. There are three columns: vowel, n (tally), prop (proportion), which lists the unique vowels in the annotation data, the number of each vowel, and its relative proportion in the data set. The data indicates, at the time of generation, there have been 376 transcriptions of word-medial unstressed [ɔ], and this accounts for 14.75% of the data set. Unstressed word-final [ɔ] (Row 3) accounts for 10.63% of the data.

vowel	n	prop
ɔ	376	14.75
ˈɐ	336	13.18
ɔ#	271	10.63
ɐ	260	10.20
...	...	...

Table 1: *Initial 4 rows of a table summarising the vowel data in the repository, generated automatically after commits are received*

The regular, *automated* derivation of summary data, allows for changes in the annotations to be closely monitored. If Table 1 were reverse-ordered on the column n, the data would show the least-frequently occurring transcriptions. From this view of the data, transcription errors become immediately obvious. For example, suppose there is single instance of 'y' in this table (i.e. a high front rounded vowel). Given the languages being analysed, it is highly likely that it is an error (perhaps a typo of the adjacent key 'u').

Additionally, the writing of code snippets for summarising the dataset assists in the data analysis process. Any future analysis requiring the extraction of vowel annotations from this data set can re-purpose the extraction parts of the scripts being used to monitor the data.

Moreover, the data extraction script need not be written in any executable language. As Git hooks are simply events that trigger a script, the scripts themselves may be in any language—be it an emuR query, a Praat or Python script—and can thus help automate repetitive tasks across a number of applications.

4. Discussion

Many of the ideas introduced and discussed are already part of the standard phonetics workflow. For example, there is usually some manual form of version control for files and the [manual] running of scripts to extract and verify data. Git as a VCS, along with hosts such as GitHub, formalises such ideas into a set of standard protocols, which allow for efficiency and transparency in the workflow—factors which become increasingly important with large-scale collaborative work.

The start up costs for implementing and using Git as a VCS are now quite minimal. This has been aided by the establishment of many cross-platform Graphical User Interface (GUI) clients: for example, GitHub Desktop, GitKraken, and SourceTree (used by all AADR and KPHON RAs). That is, regardless of the contributor's platform (Windows, OS X, Linux), everyone can use the same Git client, and become acquainted to a Git VCS workflow as a group.

Additionally, using Git as a VCS does not limit a project to a certain workflow. This fact has allowed for the training of RAs in an incremental manner. The simplest possible workflow through a Git client merely adds a number of administrative steps in the client to send and receive data. Then, additional features are introduced one by one as needed.

For KPHON and AADR, the teaching of Git/SourceTree to RAs is part of the introduction to the projects. For example, when RAs initially start any project, they must familiarise themselves with the project's annotation criteria on a practice set. For KPHON and AADR, the Git/SourceTree training is simply a part of this familiarisation.

Moreover, the formalisation of how problematic annotations and errors are communicated (e.g. via GitHub issues), in fact, facilitates the workflow, as it establishes a central repository of knowledge. For example, an RA can quickly find out if a specific problem has been encountered before by another, or past, RA, and how that problem was resolved (with details of the exact files concerned).

One should note, however, that Git and particularly its diff functionalities (i.e. summarising the changes between two versions of files) work best on non-binary data, i.e. plain text file formats (which include Praat .TextGrid, ELAN .eaf, .json). This is due to the fact that software development has depended primarily on version control of source code, and not binary assets such as image or audio files. However, as Git as a VCS is being adopted more and more by teams outside the software development tradition, more and more diff functionalities are being implemented by third parties. In short, versions of any file can be tracked through the Git VCS, however, quickly viewing what changes occured currently only works best for plain text file formats.

Similarly, automatic conflict resolution in Git works best for line-by-line mismatches in these plain-text files. This may be problematic for annotation files such as .TextGrids as their data consist of inter-related multi-line information (e.g. time + text). Such problems, however, would only emerge when a specific portion of the same tier of a given file is edited by two or more annotators at the same time. We have not encountered this as an issue in KPHON or AADR.

5. Conclusion

This paper introduced the use of a version control system (Git, in particular) in an acoustic phonetics annotation workflow. Of course, Git and other version control systems were designed with software development in mind. Consequently, it may well be discovered that—for use within acoustic phonetics—only a subset of their features are highly beneficial, while some unnecessary, and others lacking—and must be developed for the field of acoustic phonetics. For any such discovery to take place, however, gradual adoption, experimentation, and subsequent discussion of version controlled data and its role in in acoustic phonetics will be necessary.

6. References

[1] S. Cassidy, D. Estival, T. Jones, D. Burnham, and J. Burghold, "The Alveo virtual laboratory: A web based repository API," in *9th Language Resources and Evaluation Conference (LREC 2014)*, Reykjavik, Iceland, 2014.

[2] A. Rosenberg, "Rethinking the corpus: Moving towards dynamic linguistic resources." in *INTERSPEECH*, 2012, pp. 1392–1395.

[3] J. Poignant, M. Budnik, H. Bredin, C. Barras, M. Stefas, P. Bruneau, G. Adda, L. Besacier, H. Ekenel, G. Francopoulo, J. Hernando, J. Mariani, R. Morros, G. Quénot, S. Rosset, and T. Tamisier, "The CAMOMILE collaborative annotation platform for multi-modal, multilingual and multi-media documents," in *10th Language Resources and Evaluation Conference (LREC 2016)*, Portorož, Slovenia, 2016.

[4] P. Boersma, "Praat, a system for doing phonetics by computer," *Glot international*, vol. 5, no. 9/10, pp. 341–345, 2002.