MAINTAINING A GITHUB REPOSITORY

Part 2 – Setting up and managing a GitHub Project

- Lots of resources online for GitHub but they are written by **Computer Scientists who develop software**, but we are **researchers that do research.** Two main differences:
  - *Many resources online use the command line instead of GitHub Desktop*
  - *The data sets are just as important to us researchers as the code*

- Content:
  - *Setting up a new repository*
  - *Setting up a work flow for contributions*
  - *How to work with data in GitHub (GitHub & DropBox)*
  - *More tools for exploring the work already done on a repository*

Setting up a new repository

This section includes:

- User account vs. organization account ownership
- Who has access? And what type of access?
  - *Private vs. Public vs. Secret repositories*
  - *Inviting collaborators*
- Renaming and transferring repositories

Organization owned vs. user owned

- An repository is owned either by a user account or an organization account
  - *https://github.com/owner_name/repo_name*

- An organization account is special type of paid account where users with free accounts can host their repositories and get access to premium features for those repositories
  - *For example, https://github.com/worldbank*
  - *The most important premium feature on GitHub is private repositories*

- There can only be one owner, but you can transfer a repository to someone else, or to an organization

Public vs. Private - Levels of access

Read access

- *Can see the repository, create and comment on issues, and create forks*

Write access (organization owned) or Push access (user owned)

- *Read access and also commit, open and merge pull requests, and create branches*

Admin access (organization owned) or Owner (user owned)

- *Write access and change settings and move/delete repository*

Public vs. Private - Levels of access

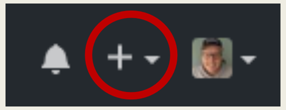
	Public repositories			Private repositories		
	Read Access	Write Access	Admin Access	Read Access	Write Access	Admin Access
User owned	All GitHub users	Users who the owner gave push access	Only the owner	Can only be granted by also giving push access	Users who the owner gave push access	Only the owner
Organization owned	All GitHub users	Users who an admin gave write access	Users who another admin gave admin access	All users within the organization (unless it is a secret repository, then admin has to give explicit read access)	Users who an admin gave write access	Users who another admin gave admin access

More on organization accounts

- Typical work flow. Organization accounts can be administered in many different ways, this is meant to be a general understanding
  - *Organization admins create repositories for a user and makes that user admin of the repository.*
  - *Only organization admins have the right to delete or transfer repositories.*
  - *Private repositories are often public within organization. You can make it private within the organization, it is called a secret repository*
  - *External user accounts can be added to private repositories in an organization*
  - *You can create teams and give those teams different level of access to your repositories*

- Organization accounts helps you market your repositories you want to share publicly
- There are no free organization accounts on GitHub

Exercise: Create a new repository

1. Click the plus sign in the top menu and select *New Repository* 
2. Pick a name, the URL will be [https://github.com/*your_username*/*repo_name*](https://github.com/your_username/repo_name)
3. Give the repository a description. For example, “This is a repository for a GitHub training”.
4. Select public repository (the only option unless you have a paid account)
5. Select *Initialize this repository with a README*
6. Do not add license or .gitignore
7. Select *Create repository*
8. Then clone this repository to your computer

Exercise: Add collaborators

1. On your repository, go to the *Settings* tab
   – *You only see this tab if you have admin access or you are the owner*
2. Go to *Collaborators* in the menu to the left.
3. Add someone that sits next to you

- Since this is a user owned repository you do not have different levels of access.
  – *You will have owner access and everyone you add will have push access (which include read access)*
- This is how you would add external collaborators to a repository owned by an organizational account, but you can assign people different level of acces.

Renaming or transferring a repository

https://github.com/*owner_name*/*repo_name*

- Rename
  - *You can change name of your repository at any time (if you have admin access)*
  - *Will change the repo_name in the URL, but the old URL will redirect visitors to the new repository.*

- Transfer
  - *You can transfer a repository to another account (if you have admin access)*
  - *The repository will be public if the new account does not allow private accounts*
  - *Will change the owner_name in the URL, but the old URL will redirect visitors to the new repository.*

- The redirect will stop working if a new repository is created for the old URL

GitHub Meta Files

This section includes:

- README.md and other files that helps user use the repository properly
- .gitignore – Prevents that private data is leaked through GitHub and is important to not slow down your repository or to make it too big.

Meta Files for collaboration

Important for our types of projects:

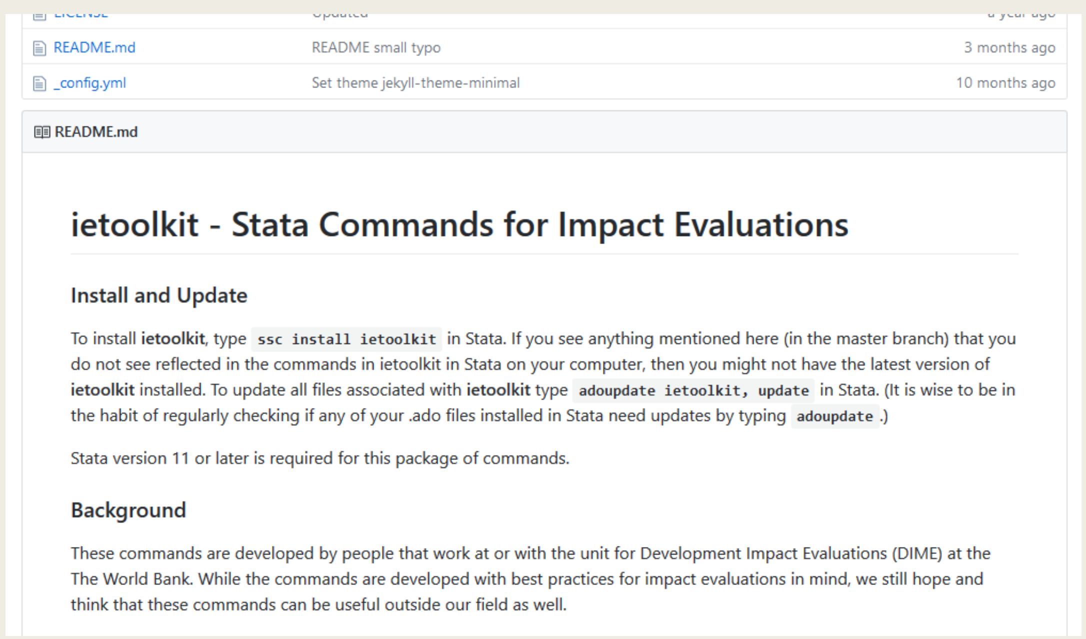
- README.md
  - *Explains the purpose of the repository and anything else useful to anyone browsing the repository*
  - *Files with exactly the name README.md are displayed in the browser when browsing to the folder that contains that file.*
  - *Common in the top folder, but works the same way in any subfolder*

Great to include, so GitHub will push for them, but only relevant to us in public repositories:

- LICENCE.md
  - *Common licenses are:*
    - MIT – the most generous. Use code however you want, credit is not needed
    - Creative Commons Attribution 4.0. Use code and adapt as you want, but give credit
- CONTRIBUTING.md
  - *Tell user how you want them to contribute to the repository*

What is Markdown?

- All meta files had the file extension .md – stands for Markdown. Markdown is a method to format text written in raw text files.
  - *Markdown is a simplified markup language. The M in HTML stands for markup, so it is a simplified version of a commonly used principle for formatting raw text*
- Markdown is not used only in .md files in GitHub. It is also used in discussion threads etc. Markdown is also used in many contexts other than GitHub, for example, in R.



README.md	README small typo	3 months ago
_config.yml	Set theme jekyll-theme-minimal	10 months ago

README.md

ietoolkit - Stata Commands for Impact Evaluations

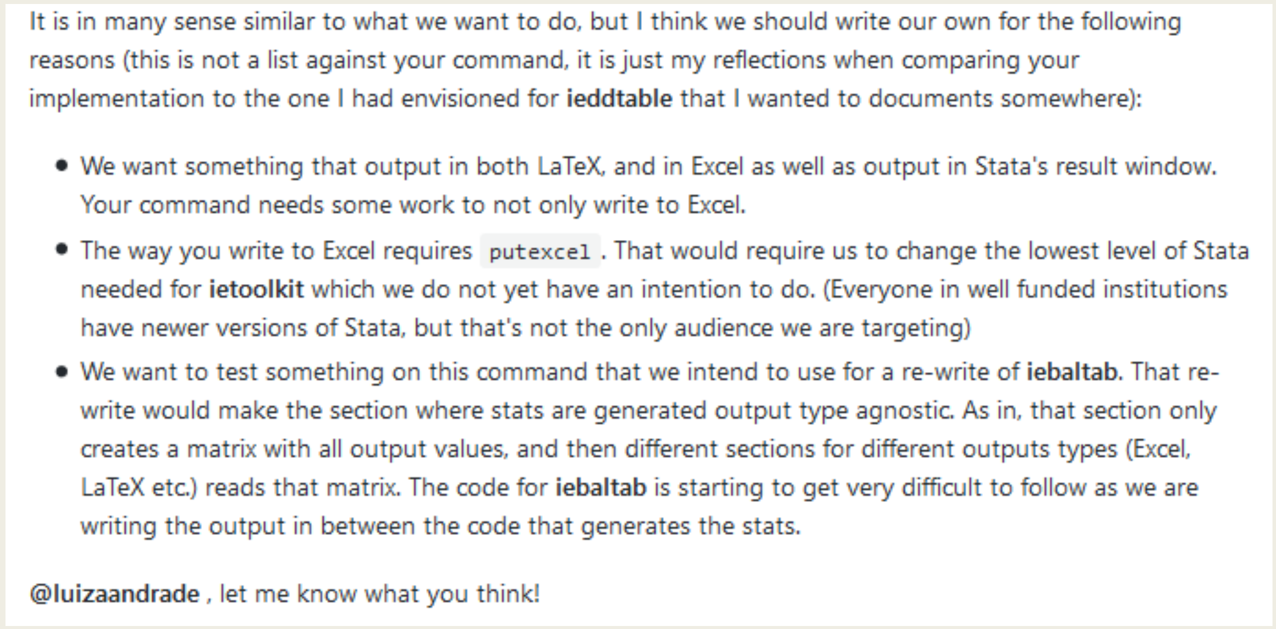
Install and Update

To install **ietoolkit**, type `ssc install ietoolkit` in Stata. If you see anything mentioned here (in the master branch) that you do not see reflected in the commands in ietoolkit in Stata on your computer, then you might not have the latest version of **ietoolkit** installed. To update all files associated with **ietoolkit** type `adoupdate ietoolkit, update` in Stata. (It is wise to be in the habit of regularly checking if any of your .ado files installed in Stata need updates by typing `adoupdate`.)

Stata version 11 or later is required for this package of commands.

Background

These commands are developed by people that work at or with the unit for Development Impact Evaluations (DIME) at the The World Bank. While the commands are developed with best practices for impact evaluations in mind, we still hope and think that these commands can be useful outside our field as well.



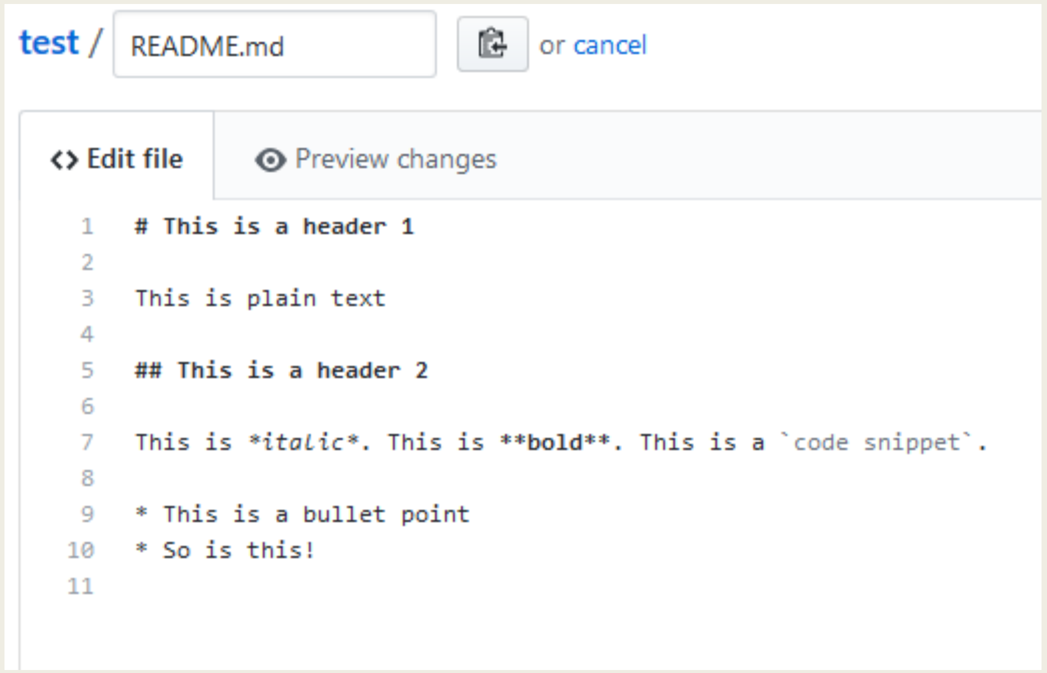
It is in many sense similar to what we want to do, but I think we should write our own for the following reasons (this is not a list against your command, it is just my reflections when comparing your implementation to the one I had envisioned for **ieddtable** that I wanted to documents somewhere):

- We want something that output in both LaTeX, and in Excel as well as output in Stata's result window. Your command needs some work to not only write to Excel.
- The way you write to Excel requires `putexcel`. That would require us to change the lowest level of Stata needed for **ietoolkit** which we do not yet have an intention to do. (Everyone in well funded institutions have newer versions of Stata, but that's not the only audience we are targeting)
- We want to test something on this command that we intend to use for a re-write of **iebaltab**. That re-write would make the section where stats are generated output type agnostic. As in, that section only creates a matrix with all output values, and then different sections for different outputs types (Excel, LaTeX etc.) reads that matrix. The code for **iebaltab** is starting to get very difficult to follow as we are writing the output in between the code that generates the stats.

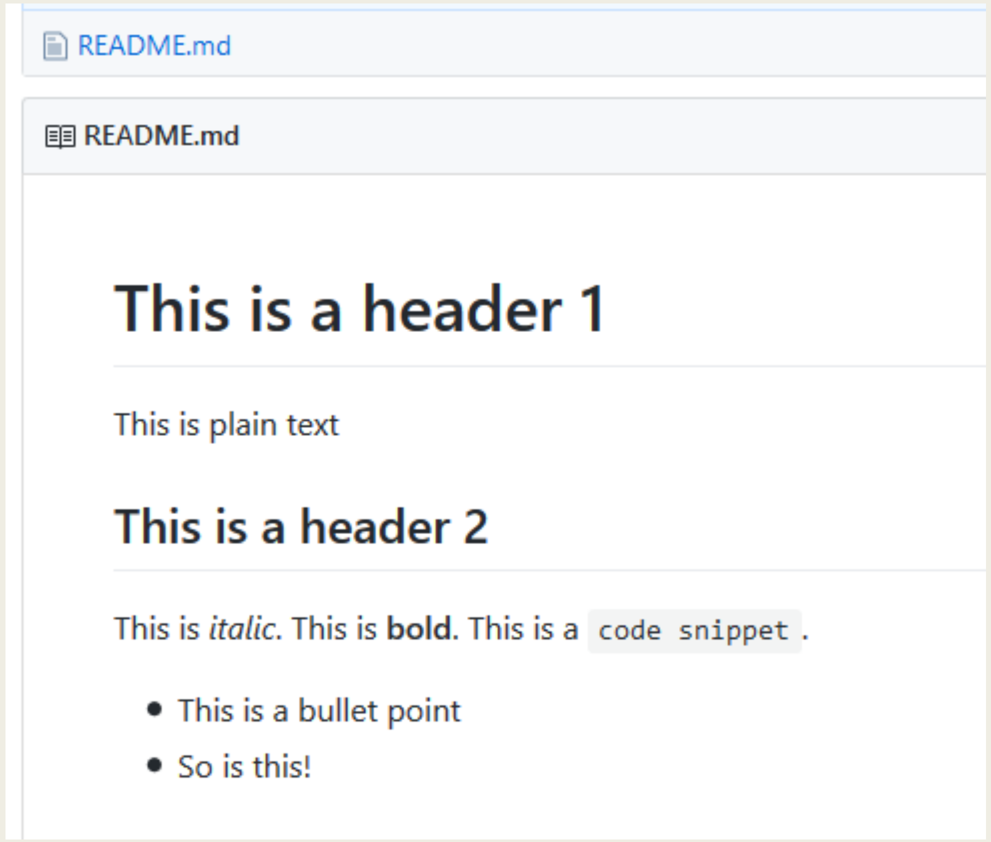
@luizaandrade , let me know what you think!

How do I use Markdown?

The whole point of markdown is that it is very simple to use. See example below. There are great resources online. Google *markdown cheat sheet* and you will find many great examples.



```
# This is a header 1

This is plain text

## This is a header 2

This is *italic*. This is **bold**. This is a `code snippet`.

* This is a bullet point
* So is this!

```



This is a header 1

This is plain text

This is a header 2

This is *italic*. This is **bold**. This is a `code snippet`.

- This is a bullet point
- So is this!

Exercise: Edit the README.md file

1. Go to the main page of your repository and click the README.md file.

2. Then click the pen icon to start editing

3. Add content to the file and use some Markdown editing.

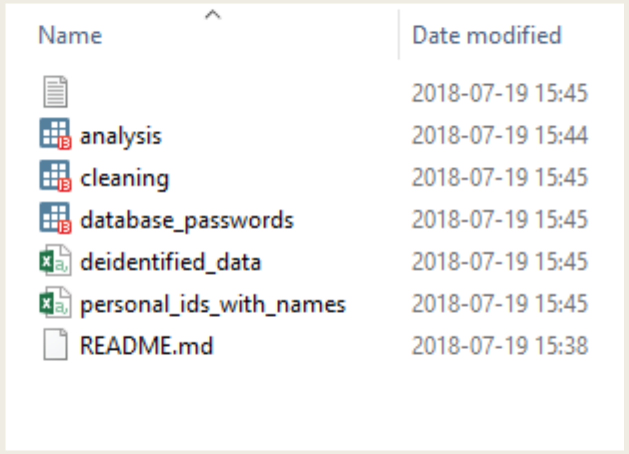
4. As with anything else in your repository, edits to meta files saved by commit the updates. See the bottom of the edit page. This put the edits to the Meta files in the same time line as all other commits, and this can be helpful as you can read the documentation of the repository at the time of any commit

.gitignore – a very important Meta file

- The .gitignore file controls which files GitHub Desktop will upload to GitHub.com. Examples of files you do not want to share are:
  - *Data files with sensitive data*
  - *Code files with information such as data base password*
  - *Binary files in general*
  - *System files and other temporary files*

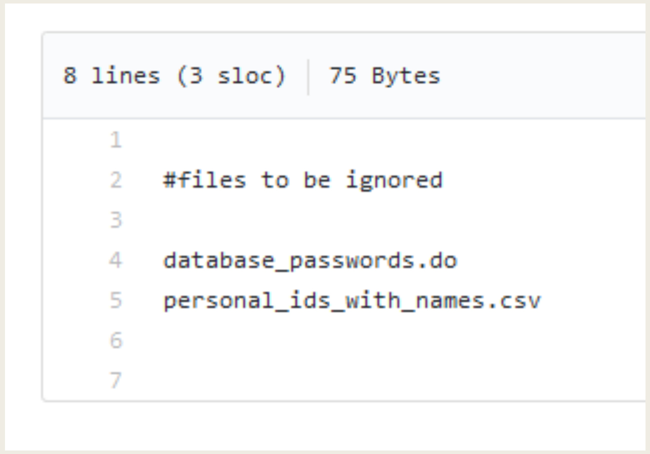
- At DIME we have developed a template that we recommend all of you to use, at least as a starting point
- We discuss how to share these types of files later

.gitignore – A basic example



Name	Date modified
	2018-07-19 15:45
analysis	2018-07-19 15:44
cleaning	2018-07-19 15:45
database_passwords	2018-07-19 15:45
deidentified_data	2018-07-19 15:45
personal_ids_with_names	2018-07-19 15:45
README.md	2018-07-19 15:38

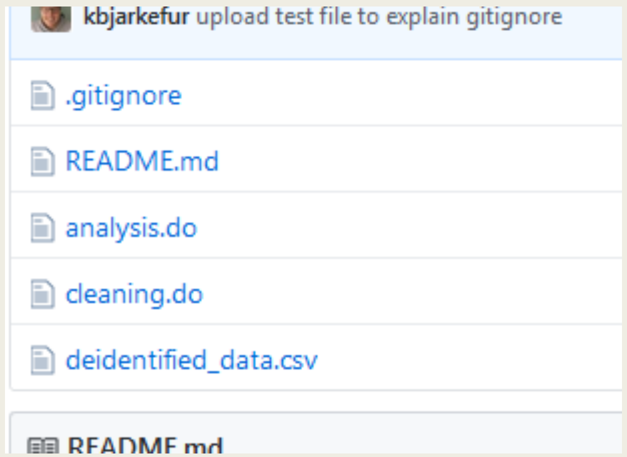
+



8 lines (3 sloc) | 75 Bytes

```

#files to be ignored

database_passwords.do
personal_ids_with_names.csv


```

=



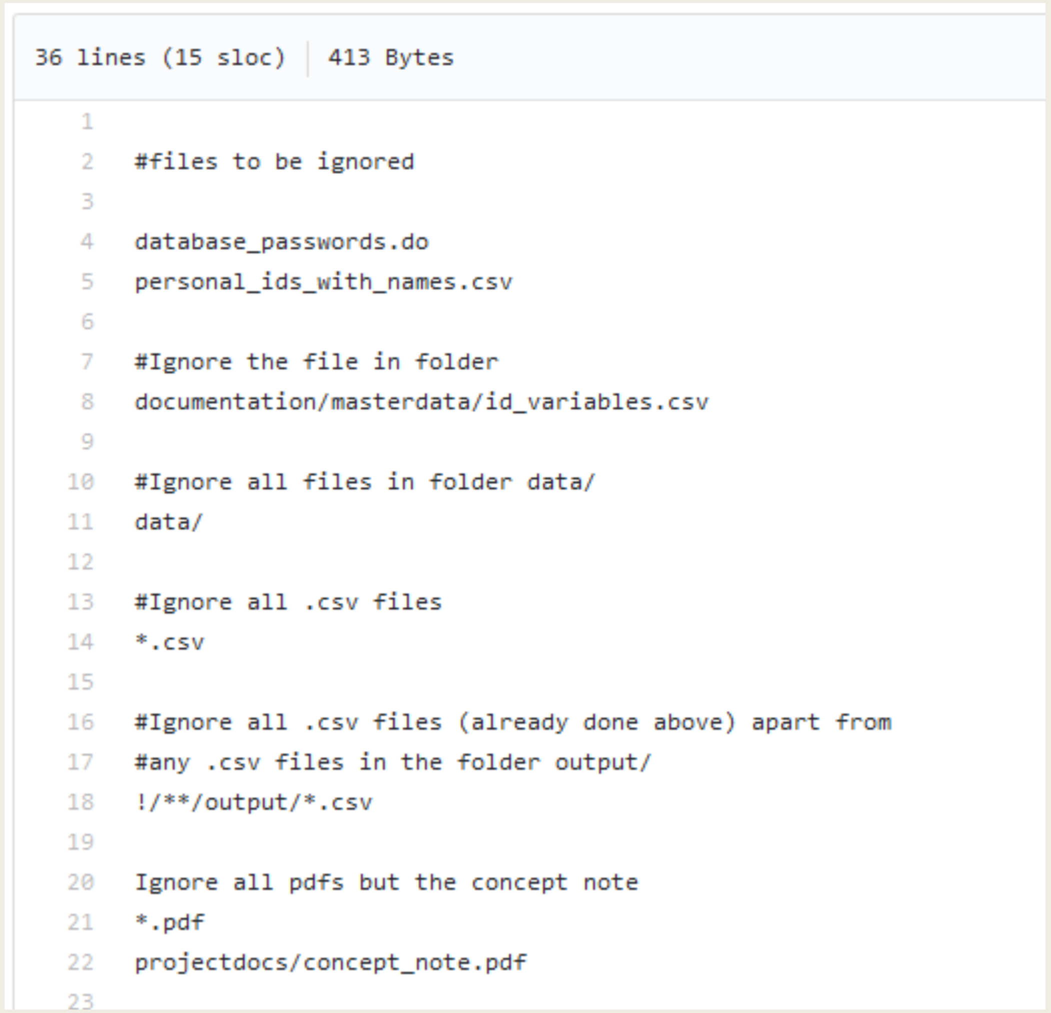
kbjarkefur upload test file to explain gitignore

.gitignore
README.md
analysis.do
cleaning.do
deidentified_data.csv

README.md

.gitignore – a more realistic example

- There are short cuts so that you do not have to add every single file you want to ignore
- You can ignore with any level of granularity
- You can make a rule and then make specific exceptions from that rule. As in ignore all csv file but once specific csv.
- Google *gitignore rules* for great resources
- Use an example, like the DIME Template as a starting point



36 lines (15 sloc) | 413 Bytes

```

#files to be ignored

database_passwords.do
personal_ids_with_names.csv

#Ignore the file in folder
documentation/masterdata/id_variables.csv

#Ignore all files in folder data/
data/

#Ignore all .csv files
*.csv

#Ignore all .csv files (already done above) apart from
#any .csv files in the folder output/
!/**/output/*.csv

Ignore all pdfs but the concept note
*.pdf
projectdocs/concept_note.pdf

```

Exercise: Create a .gitignore using DIME’s template

1. Open GitHub.com in two tabs in your browser. In one tab open your repository, in the other tab go to https://github.com/worldbank/DIMEwiki/tree/master/Topics/GitHub
2. In the GitHub folder in DIMEwiki repository, click the file **gitignore_template_general.txt.** Then click the button called *Raw* and then copy all the content in your browser by highlighting the text and click Ctrl+C (or equivialent)
3. In the top folder in your repository click the button called *Create a new file*. Give the file exactly the name *.gitignore* where the . before the word is important
4. Paste the content you copied from the template in step 2 to your file and commit this edit.
5. Then go to GitHub Desktop to sync this edit to your local clone.

This is not the only way to do this, you could, write your own ignore file, but we recommend that this is a starting point for you.

More on .gitignore files

- Ignoring specific files:
  - *If you see a file in the staging area (where you select files to commit) that should never be synced to the cloud, right click the file and select ignore file.*
  - *The file is removed from the staging area, but you need to commit the edits to the .gitignore now*

- .gitignore only affect which files are uploaded to GitHub
  - *You can manually upload on GitHub.com a file that would otherwise be ignores, For example, concept note pdf.*
  - *This file will be downloaded to all clones next time the user syncs the clone*
  - *Local edits to this file will still be ignored*
  - *Only the edit history for binary files is stored inefficiently, so there is no issue sharing files that rarely or never changes over GitHub*

Exploring a Repository

This section includes:

- Tools to browse the repository
- To be used by a manager that wants to stay up to date on a

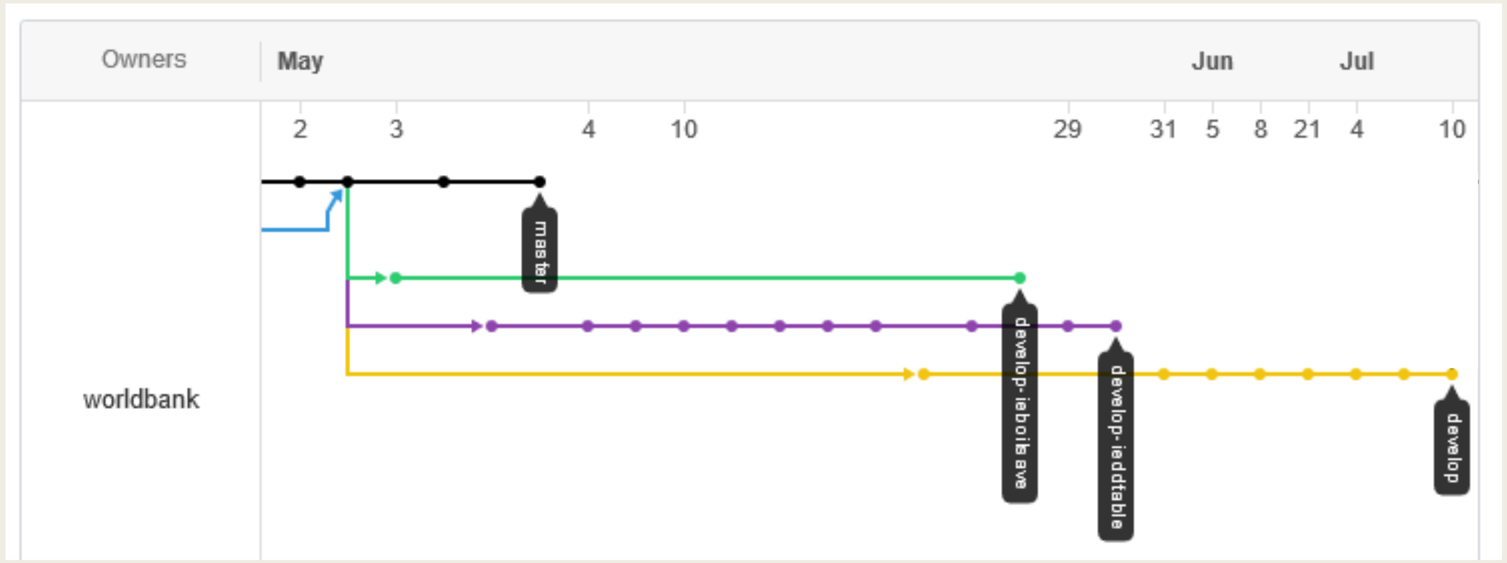
Is GitHub yet another tool that will steal my time?

- Sure, any new tool steals some time in the short run, but GitHub offers a solution to the issue that steals the most time from researchers like us
  - *Who wrote this line of code?*
  - *When did I write this line of code?*
  - *Why did I write this line of code?*
  - *GitHub provides tools that help answer those questions.*

- GitHub is meaningful even if you only do the bare minimum (no branches, hastily written commit messages, not using issues etc.), which means that GitHub won’t steal much time until you realize yourself that you save time by spending time in GitHub

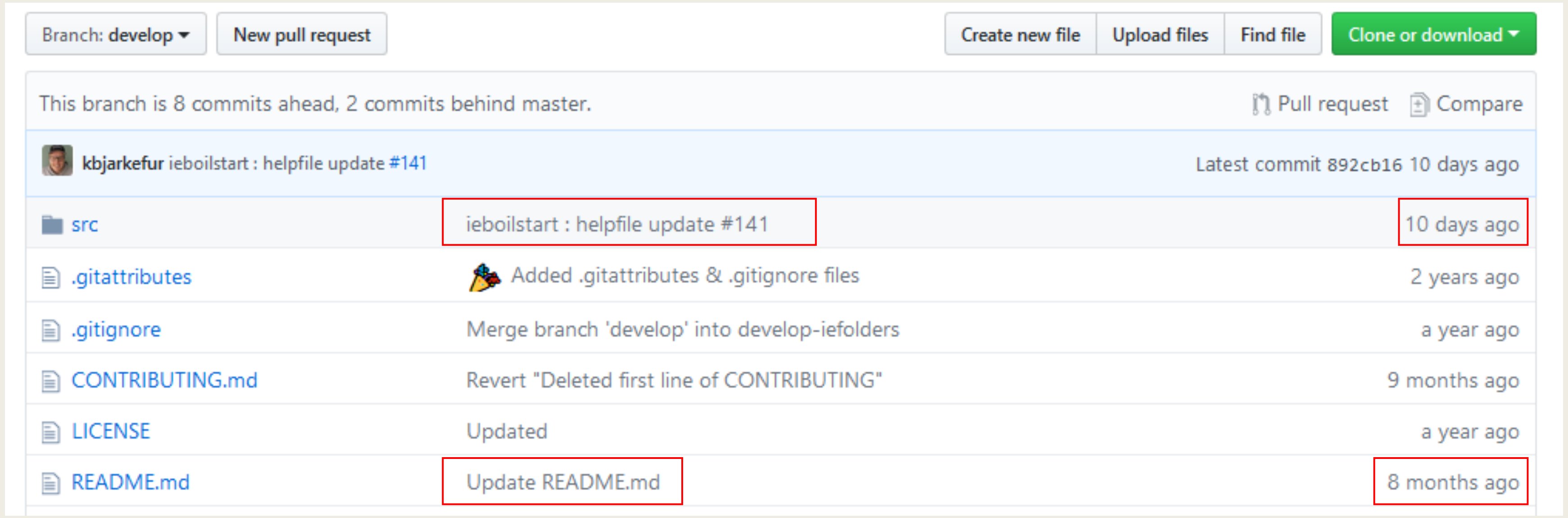
Where to start looking

- Unless you know what you are looking for, always start with the Network graph
  - *Great way to explore a new repository*
  - *Great way to get a quick idea of what your colleagues are currently working on in a repository where you are a collaborator*



Which files have been edited?

- Both a great tool to explore what has changed since last time you checked in, but also a tool to answer the question, "*This worked two months ago, what has changed since?*"
- GitHub.com displays the time and commit message to the last edit to a file, or the last time the content of a folder was edited, or the last time a branch was edited.



This branch is 8 commits ahead, 2 commits behind master.

kbjarkefur ieboilstart : helpfile update #141 — Latest commit 892cb16 10 days ago

src	ieboilstart : helpfile update #141	10 days ago
.gitattributes	Added .gitattributes & .gitignore files	2 years ago
.gitignore	Merge branch 'develop' into develop-iefolders	a year ago
CONTRIBUTING.md	Revert "Deleted first line of CONTRIBUTING"	9 months ago
LICENSE	Updated	a year ago
README.md	Update README.md	8 months ago

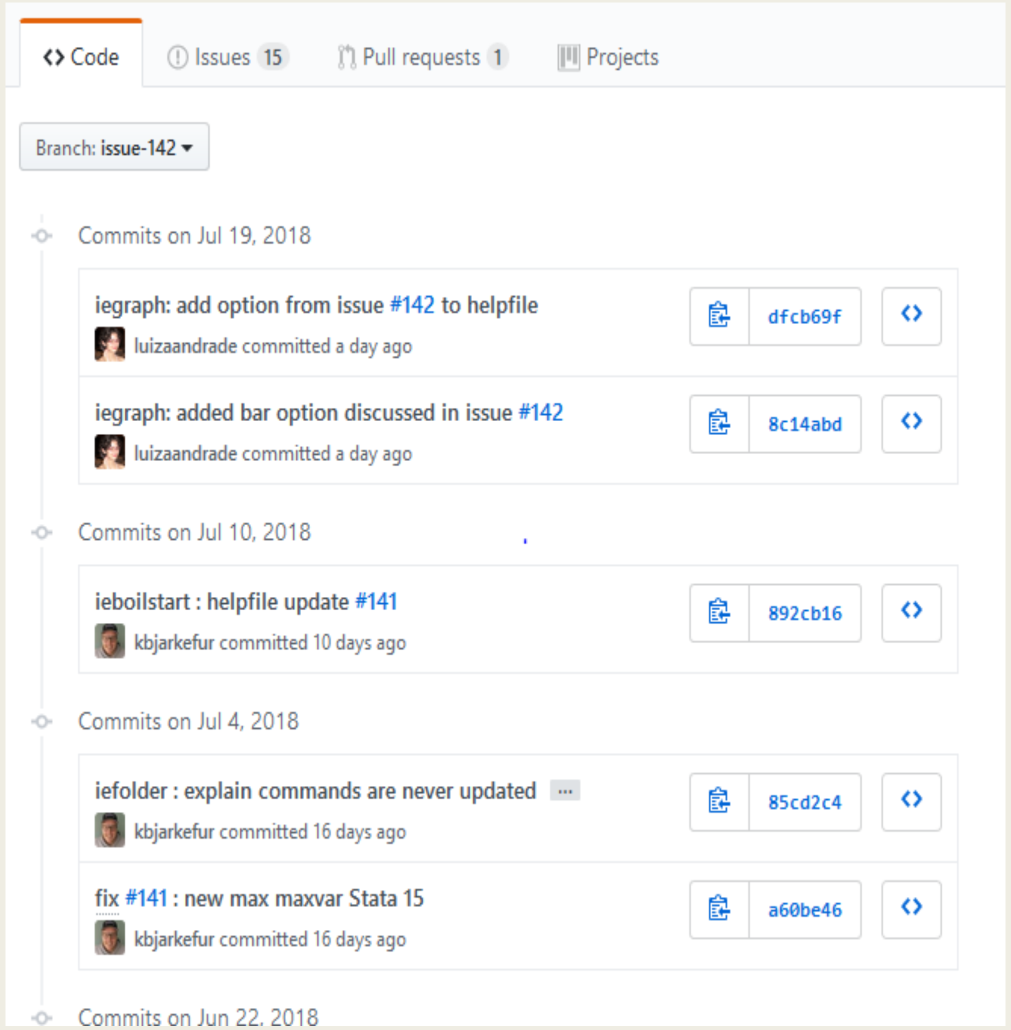
Last commit in this branch to any of the files and folders below

Last commit to edit any file in this folder

Last commit to edit this file

Commit history

- Go to the code tab and click *commits*
- You will see a complete list of all the commits that have been done to the repository
  - *Note that the list is shown for the current branch you have selected*
- You can see who made the change and when, and you can see the commit message
- You can click the commit to see all the edits made and more details about the commit
- You can click the <> to browse the repository exactly as it was at the time of that commit



Commits on Jul 19, 2018

iegraph: add option from issue #142 to helpfile
luizaandrade committed a day ago
dfcb69f

iegraph: added bar option discussed in issue #142
luizaandrade committed a day ago
8c14abd

Commits on Jul 10, 2018

ieboilstart : helpfile update #141
kbjarkefur committed 10 days ago
892cb16

Commits on Jul 4, 2018

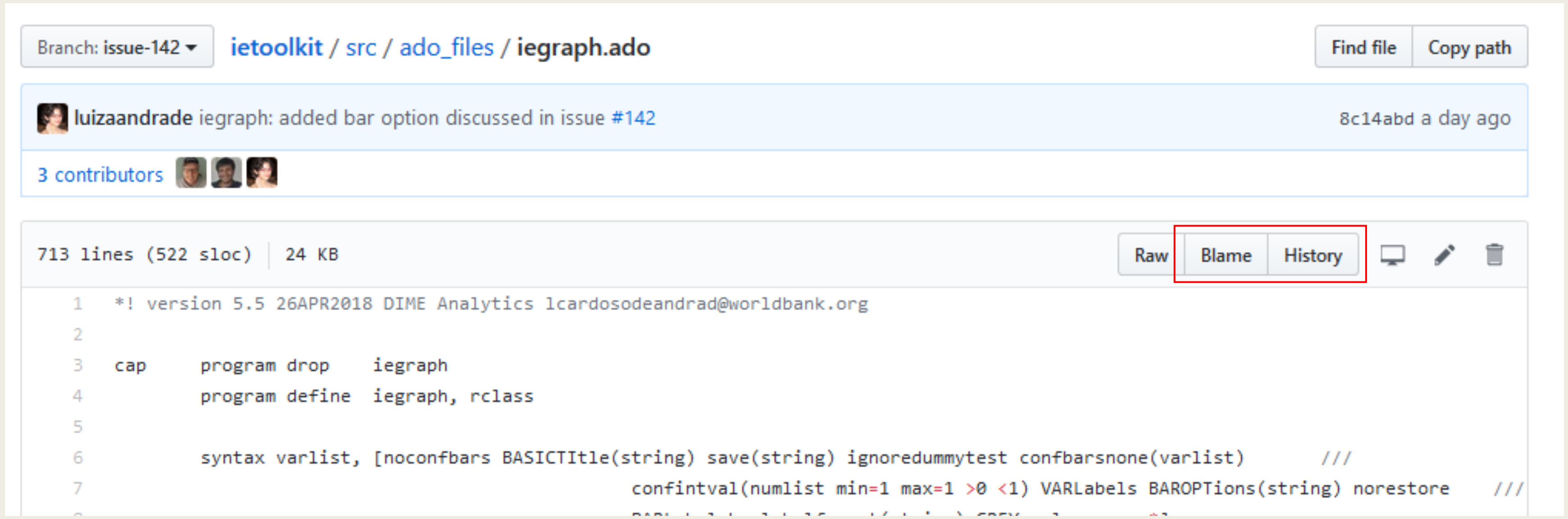
iefolder : explain commands are never updated …
kbjarkefur committed 16 days ago
85cd2c4

fix #141 : new max maxvar Stata 15
kbjarkefur committed 16 days ago
a60be46

Commits on Jun 22, 2018

Tools to explore the details

- So far the tools are for a general understanding. Once you know which file you are interested in, how do you use GitHub to understand the details of that file?
- Two great tools for this purpose. Browse to the file you are interested in and click **History** and/or **Blame**



ietoolkit / src / ado_files / iegraph.ado

luizaandrade iegraph: added bar option discussed in issue #142 8c14abd a day ago

3 contributors

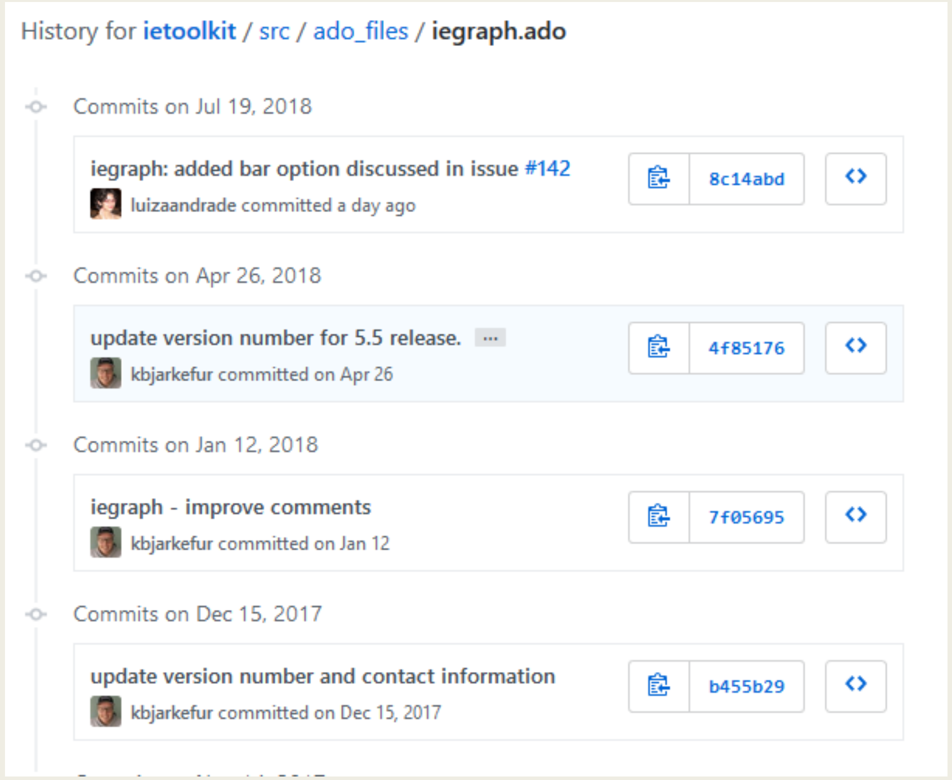
713 lines (522 sloc) 24 KB

```
*! version 5.5 26APR2018 DIME Analytics lcardosodeandrad@worldbank.org

cap     program drop    iegraph
        program define  iegraph, rclass

        syntax varlist, [noconfbars BASICTItle(string) save(string) ignoredummytest confbarsnone(varlist)      ///
                                                confintval(numlist min=1 max=1 >0 <1) VARLabels BAROPTions(string) norestore    ///
```

Tools to explore the details - History

- A list of all commits that changed anything in this in this file. Similar to the *commit history* but only for this file.
- You can see the who made the edit and when.
- Click on the commits to see the edits made in this commit. You will then also see the edits made to other files in that same commit.
- Click the <> symbol to browse the repository in the state it was at that time of the commit. Then you can, for example, browse the *commit history* at that point in time to see all earlier commits made to any file

- History can also be used on a folder. Shows the commit history for all files in that folder. Click the folder names in the file path.



History for ietoolkit / src / ado_files / iegraph.ado

Commits on Jul 19, 2018

iegraph: added bar option discussed in issue #142
luizaandrade committed a day ago
8c14abd

Commits on Apr 26, 2018

update version number for 5.5 release. ...
kbjarkefur committed on Apr 26
4f85176

Commits on Jan 12, 2018

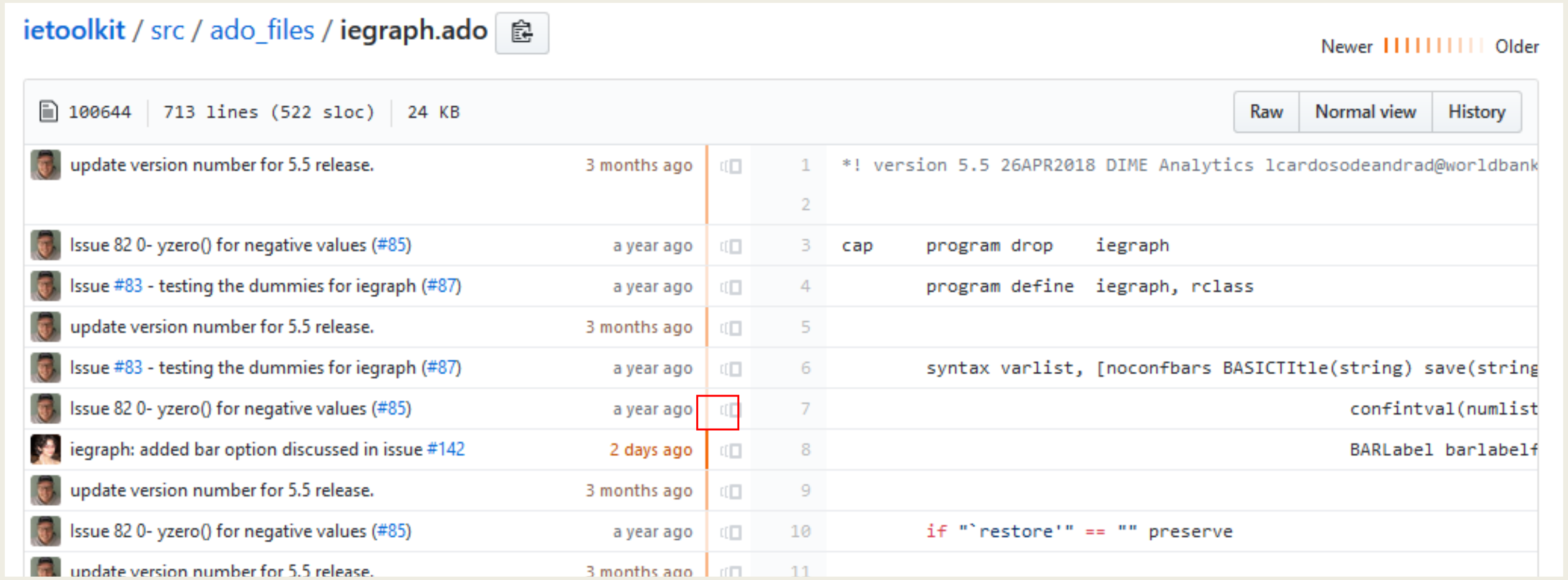
iegraph - improve comments
kbjarkefur committed on Jan 12
7f05695

Commits on Dec 15, 2017

update version number and contact information
kbjarkefur committed on Dec 15, 2017
b455b29

Tools to explore the details - Blame

- In the network graph or the commit history, you browse the edit history starting at ta point in time. In blame you start with a line of code you want to explore. For each line of code, blame shows the most recent edit to that line, who did the edit and when, and the commit message for that edit.
- By clicking the commit name you see what else was edited when that line of code was edited
- By clicking the symbol to the left of the line number (see red rectangle in image) you are shown blame previous to that commit. So you can click your way back the history of a single line of code


ietoolkit / src / ado_files / iegraph.ado

100644 | 713 lines (522 sloc) | 24 KB

update version number for 5.5 release.	3 months ago	`*! version 5.5 26APR2018 DIME Analytics lcardosodeandrad@worldbank`
Issue 82 0- yzero() for negative values (#85)	a year ago	`cap     program drop    iegraph`
Issue #83 - testing the dummies for iegraph (#87)	a year ago	`        program define  iegraph, rclass`
update version number for 5.5 release.	3 months ago	
Issue #83 - testing the dummies for iegraph (#87)	a year ago	`        syntax varlist, [noconfbars BASICTItle(string) save(string`
Issue 82 0- yzero() for negative values (#85)	a year ago	`confintval(numlist`
iegraph: added bar option discussed in issue #142	2 days ago	`BARLabel barlabelf`
update version number for 5.5 release.	3 months ago	
Issue 82 0- yzero() for negative values (#85)	a year ago	``        if "`restore'" == "" preserve``
update version number for 5.5 release.	3 months ago	

Documentation – Tags and Version

- The previous tools has all been about finding something done in the past. But what if you did something now that you know someone in the future might want to find easily?
  - *Baseline report published*
  - *Randomized treatment assignment*
  - *A research assistant who has be working on the code leaves the project*
  - *Before reorganization of the folders in the repository*
- Tags – Label a certain point of time in the repository. You can very easily find and go back in to the point in time of that label, and browse the repository and run the code as it if was that day.
- In the wider GitHub community tags are used to mark releases of software versions
- If you click a tag, then the commit the tag points to and then click *Browse files*, you get to the state of the repository at that point in time. Most tools like *Blame* and *History* works just as if you were visiting the repository the day that commit was done

Exercise - Create a tag

1. On the main repository of your repository, click where it says *releases*.
2. Then click the button that says *Create a new release.*
3. Name the release. It must be a unique name. GitHub suggests that you name it *v1.0* etc. but since it is not exactly versions in the software sense we are creating, you can give it a still short but more descriptive name, such as *baseline-report* or *Eva-left.*
4. You can tag any commit in your repository, but in the standard case you want to tag the most recent commit in the master branch, so select master.
5. Give it a slightly longer and more descriptive title
6. Write all information that could be useful to a future user in the description
   – *Describe what replication efforts were done before this tag*
   – *Link to paper/report if tag represent the point of time of a publication*
   – *Description of the types of tasks the team member who is leaving worked with*
7. Then click *Publish release*

Exercise – Explore a tag

1. Go back to the main page of your repository. See how it now says that you have one release. Click the *release* button again and see the release you just did.
2. Click the button called *Tags* to see a more condensed list of releases. This is useful when you start to have many tags. There is a small difference between tags and releases, but it only applies if we are developing software, so stick to releases and treat releases and tags as the same thing.
3. To run the code at the point of any tag, download the .zip file that contains all the content of the repository at that point of time. Save this somewhere else on your computer and run the files.

Work Flow

This section includes:

- How to best use branches
- Why is combining GitHub and DropBox complicated?
- Meta files README.md, .gitignore etc.
- Setting up and sharing a folder structure

Branches

- Branches are key to organizing a good work flow using GitHub
- Commits should not be done directly to the Master branch, make edits in other branches and merge to the Master branch.
  - *Even if it is only you working in the repo, create a branch and work only I that one and merge to master after high level tasks are completed. The merges creates nice timestamps for anyone (or yourself in the future) browsing a large segment of edits*
- How many branches should one make? How many branches are too many branches?
  - *Create a branch for each high level task. Merge, delete and re-branch regularly to reduce risk for conflicts*
  - *Branches of branches. You can branch from any commit, also from commits on branches.*

Protected Branches

- You can protect a branch so only a subset of the users with write access can edit the branch. Anyone can still submit a pull request to that branch.

- The default is that only admin users can edit a protected branch, but any user with write access can explicitly be given permission to directly edit to the protected branch.

- Protecting the main version (master branch) of the code so no one accidently edits it
  - *Everything on GitHub can be reverted, but it may make the edit history a bit cluttered*
- Each pull request can be used as a review opportunity for, for example, the PI
  - *For each pull request, GitHub provides a great dashboard overview of the changes made in that pull request. The PI can then browse the edits made to any amount of detail before approving the suggested updates.*

Exercise: Protect your master branch

1. On a repository you are admin or owner of, go to the settings tab, and select branches in the menu
2. In the *Branch protection rules* section, select the master branch.
3. Tick *Protect this branch* and then save. You do not have to worry about the other options, they are more relevant when you collaborate on a big public open source project.

- Now only owners and admin can make a commit to the master branch, or merge a pull request to the master branch
- On an organization account you can tick *Restrict who can push to this branch and* add specific users that are not admins who also can also edit the branch you just protected

Data and other binary files

This section includes:

- A few more notes on binary files
- How to share data sets in a GitHub context
- How to combine DropBox and GitHub

Raw Text Files vs. Binary Files

- Recap from part 1

	Raw Text Files	Binary Files
Examples	.txt, .do, .ado, .R, .py, .html, .csv, .tex	.doc/.docx, .dta, .xls/.xslx, .pdf
Definition	Each character (no formatting) is saved in the binary code representing that character	Segments of characters and their formatting are compressed and saved in binary code
Trade-off	Inefficient storage but a computer can read all characters without decompressing the file.	Efficient storage, especially of formatting, but needs to be decompressed to read content
GitHub implication	GitHub can detect changes in individual characters. Saving edit history is efficient, as only the characters edited is saved in each commit.	GitHub can detect that something in the file changes, but not what. Saving edit history is very inefficient as a full version of the file is saved in each commit.

Combining with DropBox

- GitHub is developed by computer scientists with their use case in mind:
  - *Computer scientist collaborate on code and apply the code to their own data*
  - *In research we share both data and code so we need a way to share this*

- Dropbox is great for sharing data and any other file type. And we have developed different best practices for how to combine the two. This section applies to other syncing services like Box, OneDrive or Google Drive, but will only mention DropBox.

Combining GitHub and DropBox

- Why don’t we simply clone the GitHub folder in the DropBox folder?
  - *Any change save in DropBox will be shared to all users. GitHub Desktop then will think that change was made on that computer*
  - *Makes it impossible to work in multiple branches.*

  This is **<u>NEVER</u>** a good idea!

- Why don’t we only use GitHub?
  - *Not all team members will know GitHub*
  - *Binary files will slow down GitHub and take up a lot of disk space*
  - *Sensitive data should not be uploaded to the GitHub cloud*

Three degrees of integrate DropBox

	No DropBox		Half DropBox		Full DropBox	
	Shared DropBox Folder	Local GitHub Clone Folder	Shared DropBox Folder	Local GitHub Clone Folder	Shared DropBox Folder	Local GitHub Clone Folder
Code	No	Yes	No	Yes	Yes	Yes
Original (raw) data	No	Yes	Yes	No	Yes	No
Intermediate/final data	No	Yes	Yes	No	Yes	No
Description	No DropBox folder. The original data is shared with or downloaded by each GitHub user that puts it in the designated folder and runs code that generate all other data sets		Separate folders. Data is saved only in DropBox, and code is saved only on GitHub. File paths point to code in the GitHub folder and to data in the DropBox folder.		Separate folders. Both code and data in DropBox, only code in GitHub. File paths in code point to data in DropBox. Edits to code is downloaded from GitHub to DropBox using command line	
Pros and cons	Great solution for some papers that start with a single data set, but that is often not the case. Need to re-run all code when changes are made by other users, therefore slow.		Requires two separate folder structures, for anyone to be able to run the code, but does not require an advanced GitHub setup.		The DropBox folder contains all files. Edits to the code in DropBox is possible. Requires an advanced set up of GitHub, but only one team members needs manage that setup.	

Method: No DropBox

- This use case is often not applicable to our type of research projects.
- Only code is shared on GitHub. Each time a repository is cloned the user have to manually put the data set in the designated folder, or the data is downloaded by the code.
- Pros:
  - *Easy to set up and follows the use case GitHub was initially intended for*
- Cons:
  - *Only feasible where only one or a few data sets are used, and that they are rarely or never updated.*
  - *Requires that all code is run by every user both when the clone is initially set up or when the data set is updated.*
- If the data is public and the project has the right to share it, it can be uploaded manually to the repository on GitHub.com and be downloaded to clones despite those file types being ignored in .gitignore

Method: Half DropBox

- One local GitHub clone folder with only code and one shared DropBox folder with everything but the code
- File paths in the code to data files point to the DropBox folder, and file paths to code files point to the cloned GitHub folder.
- Pros:
  - *Updates to data sets are shared immediately to all users through the DropBox without all users having to re-run all code each time the data set is updated.*
- Cons:
  - *Anyone running on editing the code needs access to the GitHub repository*
- The code can be copied to the DropBox folder when the project reaches milestones like publishing reports, but edits done to the code in the DropBox folder will not be tracked in the GitHub repository

Method: Full DropBox

- One local GitHub clone folder with only code and one shared DropBox folder with everything including the code
- File paths in the code to data files point to the DropBox folder, and file paths to code files point to the cloned GitHub folder.
- Pros:
  - *Updates to original and intermediate data sets are shared immediately to all users without them having to re-run the code.*
  - *The code in the DropBox folder will be up to date and accessible also to team members not using GitHub*
  - *Edits can be made to the code directly in the DropBox (although this should be avoided as things like conflicts are not as straightforward to solve this way)*
- Cons:
  - *Requires a technical one-time set-up and one manual action required each time code in DropBox needs to be updated*

Full DropBox requirements

- The team member setting up the full DropBox method must create two local clones. One non-shared personal clone and one clone in the shared DropBox folder.
- Other team members using GitHub only need to create the personal clone like in any other GitHub project
- Team members only accessing the DropBox folder do not need to worry about clones, but should be aware that editing the code directly in the DropBox folder is bad practice
- This method has easy regular updates to the cost of the set up requiring some technical steps

	Set up clone in DropBox folder	Download edits to DropBox cloud from GitHub cloud	Upload edits from DropBox cloud to GitHub cloud (no conflicts)	Upload edits from DropBox cloud to GitHub cloud (with conflicts)
How often is it required?	Only once per project	As often as you want the code in the DropBox folder to be updated	When anyone makes edits to the code directly in the DropBox folder (bad practice)	When anyone makes edits to the code directly in the DropBox folder and the code in that folder was out of date (very bad practice)
Difficulty?	Medium	Easy	Easy	Medium to Difficult

Why does Full DropBox require a technical step?

- GitHub Desktop can only handle one clone of each repository at the time and Full DropBox require two clones. By using the Command Line to interact with GitHub we can set up multiple clones.
- The Command Line is needed to get access to all features of GitHub, however, the features included in GitHub desktop covers all that we researchers need unless we want a more advanced setup, like Full DropBox.
- Most resources online use the Command Line for their solutions
  - *Most computer scientists use the Command Line to interact with GitHub as that gives them the full features of interacting with servers etc. through Git*
  - *The Command Line interface works almost exactly the same across different Git implementations, so the resources are more general that way*
- I have developed instructions for how to do Full DropBox using the Command Line here: https://github.com/kbjarkefur/GitHubDropBox

Sharing Folder Structures

- Empty folders are ignored in GitHub as they are considered to carry no information.
- When researchers set up new projects we often set up a folder structure with a template, but some of these folders will remain empty initially. For example, output folders, folders for analysis do-files, etc.
- In order to share a folder you must add a placeholder file in that folder. You can do that manually or, if you are using Stata, use the ietoolkit command – **iegitaddmd**
  - *https://dimewiki.worldbank.org/wiki/Iegitaddmd*
- **iegitaddmd** adds a README.md file in any subfolder that is currently empty, and when you commit those files all folders in your local clone is synced to the cloud

Best practices for reverting commits

- You can revert any commit in the history. However, a commit is never deleted, a revert creates a new commit that is the inverse of the commit you are reverting
- All commits can be reverted. Remember that merges, reverts, edits to ignore files etc. are also commits.
- You can only commit to the end of the commit history, this applies to reverts too. That leads to the following best practices:
  - ***Reverting the most recent commit*** *in a branch is always easy*
  - ***Reverting a commit that is the most recent commit for all the files that it modifies*** *is always easy*
  - ***Reverting a commit a few commits back into the commit history*** *is doable, but you should revert the more recent commits first. After you then have reverted the commit you initially wanted to revert, you can then re-revert the reverts of the more recent commits if you want to keep them and in the end you have only reverted the commit you initially wanted to revert*
  - ***Reverting a very old commit*** *is very likely to create conflicts difficult to solve, or requires you to revert an unreasonable large amount of more recent commits. You are probably better off editing the most recent version of the file manually to fix whatever you want to fix, and then submit that as a regular commit*

How to revert commits

- Since the best practice for reverts depends on the file history, check first where the commit sits in the commit history to confirm that reverting is your best option
  - *If it is the most recent commit in the branch, no problem to revert*
  - *If it is the most recent commit for all the files it modifies, no problem to revert*
  - *If it is only have a few commits that depend on this commit, then reverting could be a good option, but remember to revert those other commits first*
  - *If a large number of commits depends on it, then you should fix the issue manually if possible*
- In GitHub Desktop go to the History tab in the staging area (where you see the files to be committed) and right click the commit and select *Revert this commit.* This creates the inverse of the commit, and then you have to commit that revert to the cloud

Renaming and moving files

- GitHub will treat renaming or moving a file as a deletion of the file, and a creation of a new identical file with a different name or location
  - *The new file does not have the edit history of the old file. However, the commit following the renaming includes both the creation of a file with the new name, and deletion of a file with the old name. The edit history of the old file will still remain in the repository.*
- In GitHub a file is identified by its full file path from the top folder in the repository. That means that changing a folder name is treated as a renaming of all files in that folder
  - *This is treated as a new folder was created and all the content of the old folder was moved there*
- There are advanced command line tools to generate the new file by repeating all the commits of the old file to the new file. But there is no consensus in forums that the benefits of such commands justify the risks of errors associated with those commands

Make a private repository public

- You can go between private and public at any time as long as the account that owns the repository allows private accounts
- A repository cannot be half-private
  - *You cannot have private branches in a public repository*
  - *You cannot have private folders in a public repository*
  - *The full edit history is available in a public repository. So deleting private folders before making the repository public does not keep those folders private*
- You can copy the content of a private repository and publish in a new public repository if you want to publish your project without publishing all history

Documentation - Wiki

- Using the built in Wiki for documentation
  - *You can create a wiki where RAs document the code*
  - *This is not a requirement but it is often seen as convenient to have the documentation of the code at the same place as the code itself.*
  - *Also version controlled. If you change how you define something, you will know when you did so, and you will know what edits to the code was done before and which were done after*

Alternatives to GitHub

- GitHub advantages:
  - *GitHub Desktop Client, very user friendly for us who rarely use the more advanced features of Git. Most other options requires you to use the command line*
  - *Biggest user base. Good for public repositories you want to share!*
- GitHub disadvantages:
  - *Private accounts is a premium feature in GitHub. GitLab and BitBucket are a popular alternative where free accounts can create private repositories under certain conditions*
- Alternatives (google “GitHub alternatives” for many more options):
  - *BitBucket: http://bitbucket.org/ - Small organizations free, client not as simple*
  - *GitLab: http://bitbucket.org/ - Most generous free account, no client*
- Since they are all built on Git it is fairly easy to move a repository from one implementation to another

Thank You!