Description of how to make an R package and make it available on GitHub (Windows environment)

# Make an R project

To create an R package, do the following within Rstudio:

Select File -> New project

Select New Directory

Select R package using devtools (potentially need to scroll down a bit)

Select "main" directory and directory name (the latter will be a directory within the "main" directory)

Push create the directory (underscore is not allowed in the directory name)

To create an R-function within the package, type the following in the Rstudio console window

**library(devtools)**

**use\_r('survival\_demo\_data')** (or any other name for an R-function – here survival\_demo\_data)

write the needed R-function or copy it from a pre-made function

save

go to Build (upper right corner) and choose:

More -> load all

Put the cursor in the R function editor and select from the upper left

Code -> Insert Roxygen skeleton

Fill in the relevant titles and parameters etc

From Build (upper right), select

More -> Document

Repeat the above from use\_r for other function that needs to be in the project

To include a library in the project, do not use the library function but the command:

usethis::use\_package(‘PackageName’)

this command will update the DESCRIPTION file automatically such that the package will be loaded when a user is loading the current package. If you want to remove a package that is not used anymore delete the relevant line in the DESCRIPTION file (that is the standard method)

Other usethis commands that can be useful are:

usethis::use\_version()

usethis::use\_description()

usethis::use\_readme\_rmd()

usethis::use\_readme\_md()

When installing an updated version of a project, there can be R configuration problems that prevent overwriting of the old help files (occurs silently without warnings). If the help files seem not to be updated, try to uninstall the package (e.g. remove.packages("Narlal2") ), run the “Document” in the build section, and then reinstall the package – this should not be needed but is an efficient hack for this problem.

# Make an R package/library

It can be a good practice to combine the entire project into a package/library so that it can be downloaded as a single file for those that need to use the functions of the project.

The creation of a package is done with the project open within Rstudio. Go to the menu Tools -> Project options… -> Build Tools. Select Package in the "Project build tools" and set the directory to the directory of the relevant project. Click ok. Now go to the Build menu (typical upper right in Rstudio – together with Environment). Click More and then "Build source package". The zip/tar file will be created in the directory above the project directory. The zip/tar file can manually be moved into the project directory if desired.

# Description of GIT and setup from a terminal

The main idea of GIT is to keep track of the files' version and enable collaboration between people placed remotely or locally. The GIT system contains four "levels" 1) a GIT server, e.g. GITHUB on the internet or a local installation, 2) committed files on the local computer, 3) staged files on the local computer, and 4) all the files on the local computer.

The idea is that a developer is updating the local files for a given task. A given task might need to update several files. After updating the first file, the file can be "staged" (made ready for a commit). This process is repeated for the other files that need to be updated. During the development, there might be a wish to be able to roll back to a given step in the development process. When that is needed, all the staged files are committed (locally). The commit allows the developer to roll back to that time-point on the local computer. The commit process includes a description of which changes are performed for the specific commit. After evaluation of all the changes within a commit, the commit can be pushed to the central git server and is then available to all users and developers (for a simple, quick fix, all these steps might be performed as one process).

The below description assumes that GIT for windows is installed on the computer (<https://gitforwindows.org/> - use all default values during installation)

GIT can be controlled either directly from the command line (terminal) or through a GIT-GUI or using R commands from, e.g. Rstudio. The Rstudio approach might be the simplest, but it can be helpful to get a basic understanding of the structure of GIT and how to configure it using the command lines. The following is a downscaled description of GIT – for more details, see e.g. <https://happygitwithr.com/big-picture.htm> . Further down in the document, the same description is made but using Rstudio commands.

When using the command line for git it is important to know that there are several different command line editors in windows, e.g. cmd. When configuring git, it is recommended to use GIT Bash editor (windows key and type git bash – git bash is an editor provided by git for windows).

Start by opening the git bash editor

The first step is to make a local directory into a git repository. This can be done in the editor by changing the working directory to the relevant repository directory (could be as follows):

**cd C:/home/cab/my\_console\_testrepo**

All window path separators (\) shall in this editor be a forward slash (/)

Now create a local git installation in that folder using the git bash command

**git init**

This command initiates the git process and creates the .git folder of the directory, which contains all the configurations for the given repository.

An overview of git commands can e.g. be obtained from <https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet>, but the most important ones are described below.

In the editor, the configuration of git is performed using the command

**git config <followed by some instructions>**

The config command is divided into "system", "global", and "local". These levels are hierarchical (System -> global -> local) such that the "local" overwrite "global" and "global" overwrite "system". So "system" configuration will only be active if the same configurations are not defined in "global" and "local". A more precise description is that "local" inherit from "global" but overwrites those things that are defined within "local" and similar between "global" and "system".

The configuration files for each level are typically located as follows

System: C:/Program Files/Git/etc/gitconfig

Global: C:/Users/Username/.gitconfig

Local: local project folder e.g. C:/home/cab/my\_console\_testrepo/.git/config

Verification of the origin of a given item of the configuration can be done using the following command

**git config --list --show-origin**

It should be noted that the GIT bash editor does not directly support copy and paste, but it is possible to right-click and select copy and paste in the menu that appears.

The following will show a few examples of how to change some settings. If you try some of these examples, try using the command git config --list --show-origin to see how they impact the configuration. Remember that all commands can be performed at the three levels "system", "global", and "local" (so in the examples, these can be interchanged)

***git config --local user.name "CarstenBrink"*** *(define the user name at the local level)*

***git config --local --unset user.name*** *(remove the user name at the local level)*

***git config --global user.email "CarstenBrink@rsyd.dk"***

***git config --global --unset user.email***

Now try to make a new file in the git repository using the git bash editor

***Echo "This is a change made from the editor" > Readme.txt***

The status of the git repository can be obtained using the command

***git status***

If we want Readme.txt to be included in the next commit, the file needs to be staged, which can be done by the following command

***git add Readme.txt***

check e.g. the changes using git status. All the staged files can then be committed such that it is possible to make a roll back on the local computer to the situation at that commit. The commit, including a commit message, can be done as follows

**git commit -m "This is my first commit for this repository"**

The performed commits can be shown by the command

**git log**

The next step is to enable the push of the local commit to a central git server such that other people can access the files.

In the following, it is assumed that the user has made an account on GitHub, which is one of the major providers of central git repositories. A free account can be created quite easily at github.com using the sign-up function. The sign-up process will include two-factor validation based on e.g. SMS messages. The following assumes that the user has made a GitHub account. Within the web interface of the GitHub account, choose the tab named "repositories" (there might not be defined repositories yet). At the top right corner are two pull-down menus; the first is a plus sign, and the second is an icon related to the user logged in. Use the plus icon to create a new repository. Provide a repository name and description and click the "create repository" button at the bottom of the screen. The browser now shows command lines that can be used to connect to GitHub. The first example is if the user would like to obtain files from the remote repository; the second example is how to add a locale repository to GitHub (this is our situation since we have just made a local git repository above). The lines used to connect to GitHub could look like the following (maybe wait using these lines and read below)

**git remote add origin** [**https://github.com/CarstenBrink/test\_terminal.git**](https://github.com/CarstenBrink/test_terminal.git) (Set the global directory in git – this is named origin)

**git branch -M main** (rename the branch name master to main – this is the new git standard)

**git push -u origin main** (push the main branch to origin i.e. the global git repository)

if you try to use these commands within the git bash terminal, they will likely fail if the user name and password for GitHub are used. The reason is that GitHub uses two-factor login; thus, username and password are insufficient. The alternative is to use a personal access token (PAT) which can be obtained at GitHub. Within the GitHub browser, select the user's drop-down icon (upper right) and "settings". Scroll down and select "developer setting" on the left side of the screen. On the new screen, select the personal access token. Click "generate new token". Enter a descriptive text for the token and select "repo", "workflow", "ghist", and "users" (others can be selected, but these are Rstudio default values) in the tick boxes below and scroll down to the button of the page and click "generate token". The token will be shown, and it should be copied since this is the only time the token will be visible. The token obtained within an environment protected by the two factor login can now be used as a password for communicating with GitHub.

The above three lines:

**git remote add origin https://github.com/CarstenBrink/test\_terminal.git**

**git branch -M main**

**git push -u origin main**

should now be possible to use if the obtained token is utilised instead of the github password. If the first line is failing with the error "remote origin already exists", it can be because there have been tested other issues in the directory and in that case, it might be needed to start with the command

**git remote remove origin**

and then repeat the three commands stated above.

It is a bit painful to remember the token value; thus, git can utilise different credential helpers (a token is a credential). One possible way is to provide the token in the .Renvironment files in clear text (not recommended), another is to utilise that the user's home library is "secure" and provide it in clear text within that library (not recommended either). The recommended way is to use the credential manager built into windows (if you are on a windows system). Windows credential manager can be utilised by using the following command in the git bash editor

**git config --global credential.helper manager-core**

The command instructs git to cash the password the next time it is used and places it in the windows credential manager. So the next time the password/token/PAT is needed as input, it will automatically be obtained from the windows credential manager.

It is possible to see which settings are in the windows credential manager by using the windows key and typing credential manager. Select Windows Credentials. Click on the needed entry and edit or delete it as needed.

# How to use GIT within Rstudio

The first thing to validate within R is whether R has access to the installed git software (git for windows). This is done within R in the menu Tools -> Global options… -> GIT/SVN. Check that the git path is shown in the "git executable"; otherwise, browse to the correct location

Since we have predefined all the git settings, we do not need to inform Rstudio that we need to use git for version control of the project. However, there seems to be a slight bug in Rstudio; thus, we will likely need to do the following if there is no "git" tab available next to the "Environment" and "History" tabs in the upper right corner of Rstudio. In this case, go to the menu Tools -> Version control; if it is already set to git select none and repeat the process, but the second time, select git. This will create the "git" tab in the upper right corner.

Try to open one of the files within the R folder and make a change, e.g. just add a comment and save the file. The file will now appear at the git tab as modified (indicated by an M). The file can be ready for commit (staged) by checking the staged box next to the file name in the git section. This is the analogy to the manual git add command. The commit is made by pressing the commit button in the git section. Enter a commit message and push the commit button. This is the analogy to git commit -m "This is my first commit for this repository". Finally, the green arrow can be used for the push. This is the analogy of git push.

When using R in connection with git, it is a good practice to ensure that files like .Renvironment, .Rdata etc is not pushed to git since these might include sensitive patient data or local tokes. To prevent these files from being pushed to git, a .gitignore file can be used to instruct git about files not to upload to git. Within R this can be done using a one-line R command

**usethis::git\_vaccinate()**

Now git is setup and running

# How to setup GIT from within Rstudio without using command line arguments (well, almost)

The first thing to validate within R is whether R has access to the installed git software (git for windows). This is done within R in the menu Tools -> Global options… -> GIT/SVN. Check that the git path is shown in the "git executable"; otherwise, browse to the correct location. We will need two libraries in R

**library(usethis)**

**library(credentials)**

the explicit library names are not needed in the following codes but are included to indicate which package is used.

The first step is to instruct Rstudio that GIT version control should be utilised:

Tools -> Version control –> project setup

Choose git/svn section and select git for version control. After this step, the directory is now a git repository (this is the equivalence of git init in the terminal method)

As mentioned above, it is good practice to run the following R command (from within Rstudio)

**usethis::git\_vaccinate()**

Define user and email

**usethis::use\_git\_config(user.name="CarstenBrink", user.email="CarstenBrink@rsyd.dk ")**

It is assumed that a GitHub account is already set up. If not, it is described above in the terminal section how it is created.

Now we should define the origin (the directory on GitHub). An example of this is:

**usethis::use\_git\_remote(name = "origin", url="https://github.com/CarstenBrink/test\_repo.git",overwrite=TRUE)**

Inform R to use the windows credential manager system:

**credentials::credential\_helper\_set(helper='manager-core', global = FALSE)**

In the git section in Rstudio, stage one file and make a commit (see description above in the terminal section)

Now generate the needed GitHub token PAT (personal access token) using the following command

**usethis::create\_github\_token()**

The first push command needs to be performed using an editor (this is the "well almost" step). But Rstudio does have an inbuilt editor just next to the console tab. Go to the Rstudio terminal, and wait a bit to get a prompt sign ($)

In the Rstudio editor, type

**git push -u origin main**

during this first push command, the system will ask for a login either to the GitHub webpage or by use of a token (the one just obtained from the webpage). If this step results in an incorrect username or password error, it can be because an old token is stored in the windows credential manager. Since that credential is incorrect, it will automatically be deleted from the windows credential manager. So if the push command is repeated, it is likely that the second time it will ask for the token and be successful.

# How to use a GitHub package on another computer

The following three lines can be used to utilise a GitHub package within R. The first load the library devtools, the second defines the load method for windows, which should be the default but it seems not always to be the case, and the third line import the package (in this example located at: https://github.com/CarstenBrink/ TestRepo):

**library(devtools)**

**options(download.file.method = "wininet")**

**install\_github("CarstenBrink/TestRepo")**

R is now able to use all the functions within the package. A few tools to investigate loaded packages might be helpful.

To see in R which packages are loaded, use the command

**search()**

To list all function in the package named "TestRepo" use the following command:

**lsf.str("package:TestRepo")**

To list all objects in the package named "TestRepo" use the following command:

**ls("package:TestRepo")**

To get the help page of the package named "TestRepo" use the following command:

**help(package = TestRepo)**

To see the source code of a given function, e.g. "survivval\_demo\_data" use the following command:

**edit(survivval\_demo\_data)**