Statistical Rethinking and Nix

Rohit Goswami · 2020-06-07 · tools · nix · workflow · R

> *This post describes how to set up a transparent automated setup for reproducible `R` workflows using `nixpkgs`, `niv`, and `lorri`. The explanatory example used throughout the post is one of setting up the `rethinking` package and running some examples from the excellent second edition of "Statistical Rethinking" by Richard McElreath.*

Background

As detailed in an earlier post[^fn:1], I had set up Nix to work with non-CRAN packages. If the rest of this section is unclear, please refer back to the earlier post.

Setup

For the remainder of the post, we will set up a basic project structure:

```
mkdir tryRnix/
```

Now we will create a `shell.nix` as[^fn:2]:

```
# shell.nix
{ pkgs ? import <nixpkgs> { } }:
with pkgs;
let
  my-r-pkgs = rWrapper.override {
    packages = with rPackages; [
      ggplot2
      tidyverse
      tidybayes
      tidybayes.rethinking
      (buildRPackage {
        name = "rethinking";
        src = fetchFromGitHub {
          owner = "rmcelreath";
          repo = "rethinking";
          rev = "d0978c7f8b6329b94efa2014658d750ae12b1fa2";
          sha256 = "1qip6x3f6j9lmcmck6sjrj50a5azqfl6rfhp4fdj7ddabpb8n0z0";
        };
        propagatedBuildInputs = [ coda MASS mvtnorm loo shape rstan dagitty ];
      })
    ];
  };
in mkShell {
  buildInputs = with pkgs; [ git glibcLocales openssl which openssh curl wget ];
  inputsFrom = [ my-r-pkgs ];
  shellHook = ''
    mkdir -p "$(pwd)/_libs"
    export R_LIBS_USER="$(pwd)/_libs"
  '';
  GIT_SSL_CAINFO = "${cacert}/etc/ssl/certs/ca-bundle.crt";
  LOCALE_ARCHIVE = stdenv.lib.optionalString stdenv.isLinux
    "${glibcLocales}/lib/locale/locale-archive";
}
```

So we have:

```
tree tryRnix
```

tryRnix

└── shell.nix

0 directories, 1 file

Introspection

At this point:

- I was able to install packages (system and `R`) arbitrarily
- I was able to use project specific folders
- Unlike `npm`, `pipenv`, `poetry`, `conda` and friends, my system was not bloated by downloading and setting up the same packages every-time I used them in different projects

However, though this is a major step up from being chained to RStudio and my system package manager, it is still perhaps not immediately obvious how this workflow is reproducible. Admittedly, I have defined my packages in a nice functional manner; but someone else might have a different upstream channel they are tracking, and thus will have different packages. Indeed the only packages which I could be sure of were the `R` packages I built from Github, since those were tied to a hash. Finally, the setup described for each project is pretty onerous, and it is not immediately clear how to leverage fantastic tools like `direnv` for working through this.

Towards Reproducible Environments

The astute reader will have noticed that I mentioned that the `R` packages were reproducible since they were tied to a **hash**, and might reasonable argue that the entire Nix ecosystem is about **hashing** in the first place. Once we realize that, the rest is relatively simple[^fn:3].

Niv and Pinning

Niv essentially keeps track of the channel from which all the packages are installed. Setup is pretty minimal.

```
cd tryRnix/
nix-env -i niv
niv init
```

At this point, we have:

```
tree tryRnix
```

tryRnix
├── nix
│ ├── sources.json
│ └── sources.nix
└── shell.nix

1 directory, 3 files

We will have to update our `shell.nix` to use the new sources.

```
let
  sources = import ./nix/sources.nix;
  pkgs = import sources.nixpkgs { };
  stdenv = pkgs.stdenv;
  my-r-pkgs = pkgs.rWrapper.override {
    packages = with pkgs.rPackages; [
      ggplot2
      tidyverse
      tidybayes
    ];
  };
in pkgs.mkShell {
  buildInputs = with pkgs;[ git glibcLocales openssl which openssh curl wget my-r-pkgs ];
  shellHook = ''
    mkdir -p "$(pwd)/_libs"
    export R_LIBS_USER="$(pwd)/_libs"
  '';
  GIT_SSL_CAINFO = "${pkgs.cacert}/etc/ssl/certs/ca-bundle.crt";
  LOCALE_ARCHIVE = stdenv.lib.optionalString stdenv.isLinux
    "${pkgs.glibcLocales}/lib/locale/locale-archive";
}
```

We could inspect and edit these sources by hand, but it is much more convenient to simply use `niv` again when we need to update these.

```
cd tryRnix/
niv update nixpkgs -b nixpkgs-unstable
```

At this stage we have a reproducible set of packages ready to use. However it is still pretty annoying to have to go through the trouble of writing `nix-shell` and also waiting while it rebuilds when we change things.

Lorri and Direnv

In the past, I have made my admiration for `direnv` very clear (especially for `python-poetry`). However, though `direnv` does allow us to include arbitrary `bash` logic into our projects, it would be nice to have something which has some defaults for nix. Thankfully, the folks at TweagIO developed lorri to scratch that itch.

The basic setup is simple:

```
nix-env -i lorri
cd tryRnix/
lorri init
```

```
tree -a tryRnix/
```

tryRnix/

```
├── .envrc
├── nix
│   ├── sources.json
│   └── sources.nix
└── shell.nix

1 directory, 4 files
```

We can and should inspect the environment `lorri` wants us to load with `direnv` file:

```
cat tryRnix/.envrc
```

```
$(lorri direnv)
```

In and of itself that is not too descriptive, so we should run that on our own first.

```
EVALUATION_ROOT="$HOME/.cache/lorri/gc_roots/407bd4df60fbda6e3a656c39f81c03c2/gc_root/shell_gc_root"

watch_file "/run/user/1000/lorri/daemon.socket"
watch_file "$EVALUATION_ROOT"

#!/usr/bin/env bash
# ^ shebang is unused as this file is sourced, but present for editor
# integration. Note: Direnv guarantees it *will* be parsed using bash.

function punt () {
    :
}

# move "origPreHook" "preHook" "$@";;
move() {
    srcvarname=$1 # example: varname might contain the string "origPATH"
    # drop off the source variable name
    shift

    destvarname=$1 # example: destvarname might contain the string "PATH"
    # drop off the destination variable name
    shift

    # like: export origPATH="...some-value..."
    export "${@?}";

    # set $original to the contents of the variable $srcvarname
    # refers to
    eval "$destvarname=\"${!srcvarname}\""

    # mark the destvarname as exported so direnv picks it up
    # (shellcheck: we do want to export the content of destvarname!)
    # shellcheck disable=SC2163
    export "$destvarname"

    # remove the export from above, ie: export origPATH...
    unset "$srcvarname"
}

function prepend() {
    varname=$1 # example: varname might contain the string "PATH"

    # drop off the varname
    shift

    separator=$1 # example: separator would usually be the string ":"

    # drop off the separator argument, so the remaining arguments
    # are the arguments to export
    shift

    # set $original to the contents of the variable $varname
    # refers to
    original="${!varname}"

    # effectfully accept the new variable's contents
    export "${@?}";

    # re-set $varname's variable to the contents of varname's
    # reference, plus the current (updated on the export) contents.
    # however, exclude the ${separator} unless ${original} starts
    # with a value
    eval "$varname=${!varname}${original:+${separator}${original}}"
}
```

```
function append() {
    varname=$1 # example: varname might contain the string "PATH"

    # drop off the varname
    shift

    separator=$1 # example: separator would usually be the string ":"
    # drop off the separator argument, so the remaining arguments
    # are the arguments to export
    shift

    # set $original to the contents of the variable $varname
    # refers to
    original="${!varname:-}"

    # effectfully accept the new variable's contents
    export "${@?}";

    # re-set $varname's variable to the contents of varname's
    # reference, plus the current (updated on the export) contents.
    # however, exclude the ${separator} unless ${original} starts
    # with a value
    eval "$varname=${original:+${original}${separator}}${!varname}"
}

varmap() {
    if [ -f "$EVALUATION_ROOT/varmap-v1" ]; then
        # Capture the name of the variable being set
        IFS="=" read -r -a cur_varname <<< "$1"

        # With IFS='' and the `read` delimiter being '', we achieve
        # splitting on \0 bytes while also preserving leading
        # whitespace:
        #
        #    bash-3.2$ printf ' <- leading space\0bar\0baz\0' \
        #                    | (while IFS='' read -d $'\0' -r x; do echo ">$x<"; done)
        #    > <- leading space<
        #    >bar<
        #    >baz<```
        while IFS='' read -r -d '' map_instruction \
           && IFS='' read -r -d '' map_variable \
           && IFS='' read -r -d '' map_separator; do
            unset IFS

            if [ "$map_variable" == "${cur_varname[0]}" ]; then
                if [ "$map_instruction" == "append" ]; then
                    append "$map_variable" "$map_separator" "$@"
                    return
                fi
            fi
        done < "$EVALUATION_ROOT/varmap-v1"
    fi

    export "${@?}"
}

function declare() {
    if [ "$1" == "-x" ]; then shift; fi

    # Some variables require special handling.
    #
    # - punt:    don't set the variable at all
    # - prepend: take the new value, and put it before the current value.
    case "$1" in
        # vars from: https://github.com/NixOS/nix/blob/92d08c02c84be34ec0df56ed718526c382845d1a/src/nix-
build/nix-build.cc#L100
```

```
        "HOME="*) punt;;
        "USER="*) punt;;
        "LOGNAME="*) punt;;
        "DISPLAY="*) punt;;
        "PATH="*) prepend "PATH" ":" "$@";;
        "TERM="*) punt;;
        "IN_NIX_SHELL="*) punt;;
        "TZ="*) punt;;
        "PAGER="*) punt;;
        "NIX_BUILD_SHELL="*) punt;;
        "SHLVL="*) punt;;

        # vars from: https://github.com/NixOS/nix/blob/92d08c02c84be34ec0df56ed718526c382845d1a/src/nix-
build/nix-build.cc#L385
        "TEMPDIR="*) punt;;
        "TMPDIR="*) punt;;
        "TEMP="*) punt;;
        "TMP="*) punt;;

        # vars from: https://github.com/NixOS/nix/blob/92d08c02c84be34ec0df56ed718526c382845d1a/src/nix-
build/nix-build.cc#L421
        "NIX_ENFORCE_PURITY="*) punt;;

        # vars from: https://www.gnu.org/software/bash/manual/html_node/Bash-Variables.html (last checked:
2019-09-26)
        # reported in https://github.com/target/lorri/issues/153
        "OLDPWD="*) punt;;
        "PWD="*) punt;;
        "SHELL="*) punt;;

        # https://github.com/target/lorri/issues/97
        "preHook="*) punt;;
        "origPreHook="*) move "origPreHook" "preHook" "$@";;

        *) varmap "$@" ;;
    esac
}

export IN_NIX_SHELL=impure

if [ -f "$EVALUATION_ROOT/bash-export" ]; then
    # shellcheck disable=SC1090
    . "$EVALUATION_ROOT/bash-export"
elif [ -f "$EVALUATION_ROOT" ]; then
    # shellcheck disable=SC1090
    . "$EVALUATION_ROOT"
fi

unset declare

Jun 06 19:02:32.368 INFO lorri has not completed an evaluation for this project yet, expr:
$HOME/Git/Github/WebDev/Mine/haozeke.github.io/content-org/tryRnix/shell.nix
Jun 06 19:02:32.368 WARN `lorri direnv` should be executed by direnv from within an `.envrc` file, expr:
$HOME/Git/Github/WebDev/Mine/haozeke.github.io/content-org/tryRnix/shell.nix
```

Upon inspection, that seems to check out. So now we can enable this.

```
direnv allow
```

Additionally, we will need to stick to using a pure environment as much as possible to prevent unexpected situations. So we set:

```
# .envrc
eval "$(lorri direnv)"
nix-shell --run bash --pure
```

There's still a catch though. We need to have `lorri daemon` running to make sure the packages are built automatically without us having to exit the shell and re-run things. We can turn to the documentation for this. Essentially, we need to have a user-level systemd socket file and service for `lorri`.

```
# ~/.config/systemd/user/lorri.socket
[Unit]
Description=Socket for Lorri Daemon

[Socket]
ListenStream=%t/lorri/daemon.socket
RuntimeDirectory=lorri

[Install]
WantedBy=sockets.target
```

```
# ~/.config/systemd/user/lorri.service
[Unit]
Description=Lorri Daemon
Requires=lorri.socket
After=lorri.socket

[Service]
ExecStart=%h/.nix-profile/bin/lorri daemon
PrivateTmp=true
ProtectSystem=strict
ProtectHome=read-only
Restart=on-failure
```

With that we are finally ready to start working with our auto-managed, reproducible environments.

```
systemctl --user daemon-reload && \
systemctl --user enable --now lorri.socket
```

Rethinking

As promised, we will first test the setup to see that everything is working. Now is also a good time to try the `tidybayes.rethinking` package. In order to use it, we will need to define the `rethinking` package in a way so we can pass it to the `buildInputs` for `tidybayes.rethinking`. We will modify new `shell.nix` as follows:

```
# shell.nix
let
  sources = import ./nix/sources.nix;
  pkgs = import sources.nixpkgs { };
  stdenv = pkgs.stdenv;
  rethinking = with pkgs.rPackages;
    buildRPackage {
      name = "rethinking";
      src = pkgs.fetchFromGitHub {
        owner = "rmcelreath";
        repo = "rethinking";
        rev = "d0978c7f8b6329b94efa2014658d750ae12b1fa2";
        sha256 = "1qip6x3f6j9lmcmck6sjrj50a5azqfl6rfhp4fdj7ddabpb8n0z0";
      };
      propagatedBuildInputs = [ coda MASS mvtnorm loo shape rstan dagitty ];
    };
  tidybayes_rethinking = with pkgs.rPackages;
    buildRPackage {
      name = "tidybayes.rethinking";
      src = pkgs.fetchFromGitHub {
        owner = "mjskay";
        repo = "tidybayes.rethinking";
        rev = "df903c88f4f4320795a47c616eef24a690b433a4";
        sha256 = "1jl3189zdddmwm07z1mk58hcahirqrwx211ms0i1rzbx5y4zak0c";
      };
      propagatedBuildInputs =
        [ dplyr tibble rlang MASS tidybayes rethinking rstan ];
    };
  rEnv = pkgs.rWrapper.override {
    packages = with pkgs.rPackages; [
      ggplot2
      tidyverse
      tidybayes
      devtools
      modelr
      cowplot
      ggrepel
      RColorBrewer
      purrr
      forcats
      rstan
      rethinking
      tidybayes_rethinking
    ];
  };
in pkgs.mkShell {
  buildInputs = with pkgs; [ git glibcLocales which ];
  inputsFrom = [ rEnv ];
  shellHook = ''
    mkdir -p "$(pwd)/_libs"
    export R_LIBS_USER="$(pwd)/_libs"
  '';
  GIT_SSL_CAINFO = "${pkgs.cacert}/etc/ssl/certs/ca-bundle.crt";
  LOCALE_ARCHIVE = stdenv.lib.optionalString stdenv.isLinux
    "${pkgs.glibcLocales}/lib/locale/locale-archive";
}
```

The main thing to note here is that we need the output of the derivation we create here, i.e. we need to use `inputsFrom` and NOT `buildInputs` for `rEnv`.

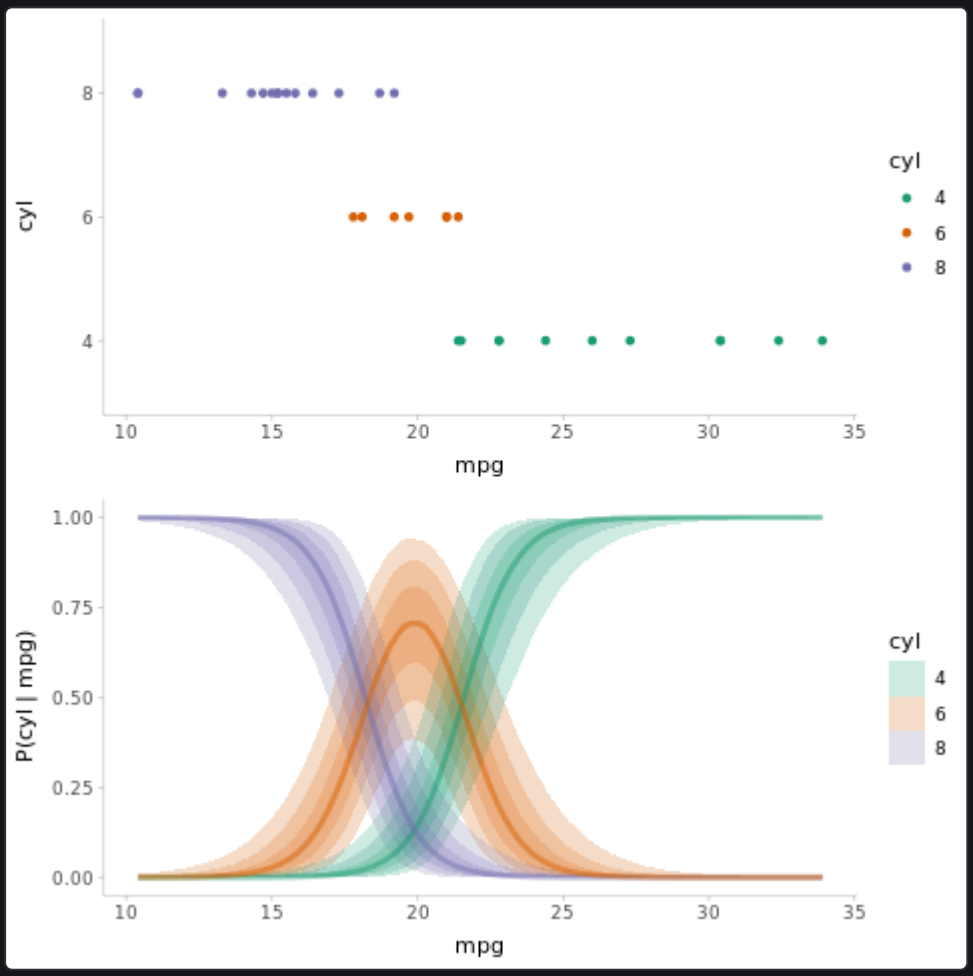
Let us try to get a nice graphic for the conclusion.

```
library(magrittr)
library(dplyr)
library(purrr)
library(forcats)
library(tidyr)
library(modelr)
library(tidybayes)
library(tidybayes.rethinking)
library(ggplot2)
library(cowplot)
library(rstan)
library(rethinking)
library(ggrepel)
library(RColorBrewer)

theme_set(theme_tidybayes())
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())

set.seed(5)
n = 10
n_condition = 5
ABC =
  tibble(
    condition = factor(rep(c("A","B","C","D","E"), n)),
    response = rnorm(n * 5, c(0,1,2,1,-1), 0.5)
  )

mtcars_clean = mtcars %>%
  mutate(cyl = factor(cyl))

m_cyl = ulam(alist(
    cyl ~ dordlogit(phi, cutpoint),
    phi <- b_mpg*mpg,
    b_mpg ~ student_t(3, 0, 10),
    cutpoint ~ student_t(3, 0, 10)
  ),
  data = mtcars_clean,
  chains = 4,
  cores = parallel::detectCores(),
  iter = 2000
)

cutpoints = m_cyl %>%
  recover_types(mtcars_clean) %>%
  spread_draws(cutpoint[cyl])

# define the last cutpoint
last_cutpoint = tibble(
  .draw = 1:max(cutpoints$.draw),
  cyl = "8",
  cutpoint = Inf
)

cutpoints = bind_rows(cutpoints, last_cutpoint) %>%
  # define the previous cutpoint (cutpoint_{j-1})
  group_by(.draw) %>%
  arrange(cyl) %>%
  mutate(prev_cutpoint = lag(cutpoint, default = -Inf))

fitted_cyl_probs = mtcars_clean %>%
  data_grid(mpg = seq_range(mpg, n = 101)) %>%
  add_fitted_draws(m_cyl) %>%
  inner_join(cutpoints, by = ".draw") %>%
  mutate(`P(cyl | mpg)` =
```

```
    # this part is logit^-1(cutpoint_j - beta*x) - logit^-1(cutpoint_{j-1} - beta*x)
    plogis(cutpoint - .value) - plogis(prev_cutpoint - .value)
  )

data_plot = mtcars_clean %>%
  ggplot(aes(x = mpg, y = cyl, color = cyl)) +
  geom_point() +
  scale_color_brewer(palette = "Dark2", name = "cyl")

fit_plot = fitted_cyl_probs %>%
  ggplot(aes(x = mpg, y = `P(cyl | mpg)`, color = cyl)) +
  stat_lineribbon(aes(fill = cyl), alpha = 1/5) +
  scale_color_brewer(palette = "Dark2") +
  scale_fill_brewer(palette = "Dark2")

png(filename="../images/rethinking.png")
plot_grid(ncol = 1, align = "v",
  data_plot,
  fit_plot
)
dev.off
```

Finally we will run this in our environment.

```
Rscript tesPlot.R
```



Conclusions

This post was really more of an exploratory follow up to the previous post, and does not really work in isolation. Then again, at this point everything seems to have worked out well. `R` with Nix has finally become a truly viable combination for any and every analysis under the sun. Some parts of the workflow are still a bit janky, but will probably resolve themselves over time.

Update: There is a final part detailing automated ways of reloading the system configuration

---