verdepcheck

Pawel Rucki

July 20, 2023

Agenda

1. Motivation
2. Solution
3. How to use?
4. How it works?
5. Discussion
6. Future work

Motivation

```
Type: Package
Package: foo.package
Version: 1.2.3.9000
Depends:
    R (>= 3.6)
Imports:
    dplyr,
    (...)
```

The above specification indicates compatibility with **all** versions of `dplyr` which is very unlikely to be true.

Motivation

```
Error: object ‘foo’ is not exported by 'namespace:bar'
```

```
`foo()` was deprecated in bar 1.0.0.
i Please use `baz()` instead.
```

The new versions of dependencies might introduce a breaking changes.

Motivation

Many interdependent packages actively developed in parallel - when to increase dependency version requirement?
Blindly increase all requirements might be incorrect as one package might be fully compatible with the older version of the other one.
Keeping all unchanged is incorrect as there might be some *breaking* changes.

Motivation

R environments for clinical trials are usually not updated very frequently.
Users might use `renv` for environment management and packages used might not get updates for some time.
As a result, your package might be used with *older* version of dependent packages.

Solution

`verdepcheck` (**ver**sion of **dep**endencies **check**) - a tool for package developers to check your package against different set of dependencies. It is meant to be used primarily in CI.
Available at:

- **`insightsengineering/verdepcheck`**
- **`insightsengineering/r-verdepcheck-action`**

Important

Do not confuse `verdepcheck` with `revdepcheck` for reverse dependency checks. Those are totally different tools!

What it is not about:

- This does not perform matrix testing of *all* versions of dependent packages.
- This does not perform search for optimal set of dependencies. It is more a *validator* of dependencies specification.
- This only looks into package versions - in particular: it does not check R versions or system architectures.

Let's call a "different set of dependencies" a *strategy*. Currently there are four *strategies* implemented:

- **max** - use *development* version of dependencies
- **release** - use *released* version of dependencies (e.g. from CRAN)
- **min_cohort** and **min_isolated** - use *minimal* version of dependencies as per `DESCRIPTION` file.

This tool is looking into a **direct** dependencies only.

What are the benefits?

- It forces you to specify minimal version of package dependencies to prevent incorrect package usage.
- Incorporate information about upcoming breaking changes.

What's needed?

For **max** strategy the algorithm will look for dependent packages references in a new section `Config/Needs/verdepcheck` of the `DESCRIPTION` file.
For **release** and **min** it's not needed (unless non-CRAN packages).

Note

This might indicate some duplication with the `Remotes` section but it cannot be used because of side effects, i.e. configuration of `verdepcheck` should have no impact package installation.

How to use?

`<strategy>_deps_check(path)`

```
library(verdepcheck)

max_deps_check("/path/to/package")
release_deps_check("/path/to/package")
min_deps_check("/path/to/package")
```

How to use?

Multiple steps:

```
library(verdepcheck)

x <- new_max_deps_installation_proposal("/path/to/package")

# resolve dependencies
solve_ip(x)

# download dependencies
download_ip(x)

# install
install_ip(x)

# run R CMD CHECK
check_ip(x)
```

How to use the output?

The main `<strategy>_deps_check` function returns a list of two elements:

- `"ip"` - installation proposal object from `pkgdepends` - see **docs**
- `"check"` - returned value from `rcmdcheck::rcmdcheck()` - see **docs**

```
x <- max_deps_check("/path/to/package")

# show dependency resolution
x$ip$show_solution()
x$ip$draw()

# create artifact
x$ip$create_lockfile("/path/to/pkg.lock")

# print R CMD CHECK results
x$check$session_info
x$check$status
```

How it works?

This package is heavily based on **`pkgdepends`** for dependency detection and resolution and also **`rcmdcheck`** for executing `"R CMD CHECK"`.
It also uses other packages like `pkgcache`, `pkgbuild`, `desc`, `cli`, `gh`.

Please also see: **r-lib/rcmdcheck/issues/195** and **r-lib/pkgdepends/issues/305**.
Workarounds have been implemented.

The algorithm

1. Read package dependencies (alongside minimal versions) from `DESCRIPTION` file using Depends, Imports, Suggests sections.
2. Derive package reference according to the strategy used.
3. Resolve dependency tree.
4. Download package sources.
5. Build and install into temporary directory.
6. Execute `"R CMD CHECK"` using directory from the previous step as a library path.

Package reference format

We are using `pkgdepends` (which is then calling `pak`) for package installation. It uses specific format for **package references**.
Few examples:

```
dplyr
dplyr@1.0.0
cran::dplyr

foo/bar
foo/bar@12ab3456
github::foo/bar

bioc::S4Vectors

/absolute/path/to/package/dir
local::.

deps::.
```

Find package references - the algorithm

- **max** (`get_ref_max()`)
  Use reference provided in `Config/Needs/verdepcheck` as is.
- **release** (`get_ref_release()`)
  For CRAN pkg: use standard reference (i.e. package name only).
  For GitHub pkg: find the latest release reference.
- **min_cohort** (`get_ref_min()`, `get_release_date()`)
  For each directly dependent package:
  - Find the lowest possible version for which version condition is met. Take the earliest possible if no condition. Use CRAN archive and GH releases or tags if no releases.
  - Derive release (commit) date.

  Calculate maximal release date and use it as PPM snapshot when resolving dependency tree.
- **min_isolated** (`get_ref_min()`, `get_release_date()`)

Similarly to the above but resolve each of the directly dependent packages separately using its own release date as PPM snapshot date. Next, combine all resolutions and resolve once again (aggregating by max version).

Discussion on the max strategy

By definition, it is using non-stable, development package version.
The result is not *stable* - it utilizes a dynamic type of reference which might result in different results each time you run this function.
For packages that does not practice version increase on each commit, package version number is not a valid package state identifier. It might happen that multiple package states will be `v1.2.3` each. Use md5 sum or package source reference instead.
This shouldn't be an obligatory type of check. It's failure should be incorporated as a notification about upcoming dependent breaking changes.

Discussion on the release strategy

By definition, it is using a stable package version.
Similarly to the previous one, the result is not *stable* - it utilizes a dynamic type of reference which might result in different results each time you run this function.

Discussion on the min strategy

It is quite complex topic how to set up an environment with packages that respects minimal version requirements and also are:

- **installable**
- **resolvable** - no conflicts when resolving joint dependency tree
- **coherent** - each package fully compatible with its dependencies. In practice it means that a package cannot be released earlier than any of its dependency.

This package does not aim to find an optimal minimal environment but it’s rather a validator of existing dependency specification.

Discussion on the min strategy

min_cohort fulfills all of above criteria but it doesn't necessarily mean it's the optimum (i.e. minimal possible).
The coherence attribute is debatable as many user errors we have encountered stems from non coherent environment. Yet another question to what incoherency extend should we allow the target environment to be. **min_isolated** strategy propose one option by joining individual, self-coherent trees which after joining doesn't have to be coherent anymore.

Real-life observations:

- Some very old packages are not compilable on modern machines and you have to increase versions to make it installable.
- The package resolution is just the first step. The next one is to use them in `R CMD CHECK` and validate its compatibility.
- In order to enter GHA debug lines, `verdepcheck` will continue on error entering `R CMD CHECK` step when e.g. dependency resolve failed. If you encounter `vignette builder 'knitr' not found` error - most likely you failed on resolve or install.

When to run it?

As discussed earlier, most of the tests are not *stable*. Therefore, this test should be triggered periodically and not on code change (unless changes includes dependencies and its usage). Triggering it on change might give a false message that a failure stems from code changes which doesn't have to be true.

Future work:

- Idea: add `branch` strategy with recursive resolve for possible replacement of `staged.dependencies`.
- Idea: add custom strategies.
- Idea: add recursive version lookups.
- Idea: new functionality (probably a separate package) that gets you optimal minimal dependency specification / environment.
- Enhance `r-verdepcheck-action` output readability.
- Remove workarounds implemented once referenced issues are closed.

THANK YOU!

Error