Documentation

Contents

Installation

Before anything, make sure the *DependencyReviewer* package is installed.

remotes

The latest version is usually available on GitHub, and is installable with the *remotes* package.

```
# If you do not have remotes installed:
install.packages("remotes")

# Install DependencyReviewer with remotes:
remotes::install_github("darwin-eu/DependencyReviewer")
```

install.packages

DependencyReviewer 1.0.0 is also available on CRAN, and can be installed using `install.packages` as well.

```
install.packages("DependencyReviewer")
```

Usage

```
library(DependencyReviewer)

# Other packages that are used in the examples
library(DT)
library(ggplot2)
library(dplyr)
library(igraph)
library(GGally)
```

getDefaultPermittedPackages

What does it do? The `getDefaultPermittedPackages` function retrieves a list of packages from several on,- and offline data sources. These data sources include:

1. Base packages with a high priority `installed.packages(lib.loc = .Library, priority = "high")`
2. Tidyverse packages
3. OHDSI/HADES packages
4. Packages hosted on the *DependencyReviewerWhitelists* repository
5. Finally the function will also retrieve the defined packages' dependencies recursively, and add them to the list.

These packages are deemed *OK* to use. This list will, and should change overtime as packages become outdated, get replaced, or added to the list.

What does it need? `getDefaultPermittedPackages` does not require any arguments.

What does it return? `getDefaultPermittedPackages` returns a class of data.frame with columns: *package* and *version*

```
datatable(getDefaultPermittedPackages())
#>  Updated metadata database: 4.91 MB in 12 files.  Updated metadata database: 4.91 MB in 12 files.
#>
#>  Updating metadata database  Updating metadata database  Updating metadata database ... done  Updatin
#>
#> Writing temp file
#> PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, pleas
```

Show 10 entries Search:

	package	version
1	KernSmooth	4.2.2
2	MASS	4.2.2
3	base	4.2.2
4	boot	4.2.2
5	class	4.2.2
6	cluster	4.2.2
7	compiler	4.2.2
8	datasets	4.2.2
9	foreign	4.2.2
10	grDevices	4.2.2

Showing 1 to 10 of 242 entries

Previous 1 2 3 4 5 … 25 Next

checkDependencies

What does it do? Now that we have defined our 'whitelisted' packages, `checkDependencies` allows us to check our currently used dependencies against it. `checkDependencies` will run `getDefaultPermittedPackages` internally so there is no need to run the two separately to check your dependencies against the white list.

What does it need? `checkDependencies` has two optional arguments:

1. **packageName** default (`NULL`): Expects a character string of a package name. Example: *"ggplot2"*.
2. **dependencyType** default (`c("Imports", "Depends")`): Expects a character vector of at least length 1 of dependency types. The supported types are: *"Imports"*, *"Depends"*, and *"Suggests"*.

Because both arguments are optional it can also be run without specifying anything. The function will then assume that it is run **inside** a package-project environment. This is specifically useful when working on, or reviewing a package.

What does it return? `checkDependencies` prints out a message in the console that informs the user if all their used package dependencies are whitelisted or not. If not it instructs the user where to go to request the packages to be whitelisted.

```r
# Assumes the current environment is a package-project
# Defaults are: packageName = NULL, packageTypes = c("Imports", "Depends")
checkDependencies()

# Check dependencies for installed package "dplyr"
checkDependencies(
  packageName = "dplyr"
)
```

1. If packages are not approved yet:

```r
# Check Imports and Suggests
checkDependencies(
  packageName = "dplyr",
  dependencyType = c("Imports", "Suggests")
)
#> Get from temp file
#>
#> -- Checking if packages in Imports and Suggests have been approved --
#>
#> ! Found 9 packages in Imports and Suggests that are not
#> approved
#> >   1) Lahman
#> >   2) RMySQL
#> >   3) RPostgreSQL
#> >   4) bench
#> >   5) covr
#> >   6) lobstr
#> >   7) microbenchmark
#> >   8) nycflights13
#> >   9) testthat
#> ! Please create a new issue at https://github.com/mvankessel-EMC/DependencyReviewerWhitelists/ to re
#> > |package |version |date | downloads_last_month|license |url |
#> |:-------|:-------|:----|--------------------:|:-------|:---|
```

As you can see, it returns a list of all the packages that are not white listed. Below the list it will display some information in a *markdown table* format. This will come in handy later on. The table has six columns: **1**) package, **2**) version, **3**) date, **4**) downloads_last_month, **5**) license, and **6**) url.

Note that only packages available on CRAN are reported in the table. Non-CRAN packages will still show up in the list.

2. If all packages are approved:

```r
# Only check directly imported dependencies of installed package "dplyr"
checkDependencies(
  packageName = "dplyr",
  dependencyType = c("Imports")
)
#> Get from temp file
#>
#> -- Checking if package in Imports have been approved --
#>
#> v All package in Imports are  already approved
```

Notice how *"Imports"* and *"Depends"* packages of dplyr are whitelisted, but *"Suggests"* packages are not.

Requesting packages to be whitelisted

If you find that some packages are not yet whitelisted, you can request them to be. The DependencyReviewerWhitelists repository on GitHub houses the white list for *DependencyReviewer*.

To request new packages a new issue can be created on this repository.

Assuming we have the following output from `checkDependencies`:

```
Get from temp file

  Checking if packages in Imports and Suggests have been approved

! Found 3 packages in Imports and Suggests that are not
approved
→   1) GGally
→   2) lintr
→   3) pak
! Please create a new issue at https://github.com/mvankessel-EMC/DependencyReviewerWhitelists/ to reques
|package |version |date                | downloads_last_month|license            |url
|:-------|:-------|:-------------------|--------------------:|:------------------|:---------------------
|GGally  |2.1.2   |2021-06-21 03:40:10 |                86657|GPL (>= 2.0)       |https://ggobi.github.
|lintr   |3.0.2   |2022-10-19 08:52:37 |                61729|MIT + file LICENSE |https://github.com/r-
|pak     |0.3.1   |2022-09-08 20:30:02 |                39420|GPL-3              |https://pak.r-lib.org/
```

When creating a new issue, a request template is available.

Figure 1: Request template button

This template asks for some basic information about the requested packages, and a reason as to why the requested packages should be whitelisted.

Initially it displays some dummy information as to what a request might look like.

Firstly it asks us is to supply a table in markdown format with some basic information about the packages. We can copy this from the output message from the `checkDependencies` function.

Then it asks us to give a description as to why we would like these packages to be whitelisted.

Finally, we can add some additional information if required.

We can then preview our request issue:

If everything looks good, we can submit the issue.

summariseFunctionUse

What does it do? `summariseFunctionUse` goes through all specified R-files and attempts to list all the functions used in those files. It will also report in what file the function was found, at what line number the function call was found, and from which package the function comes from.

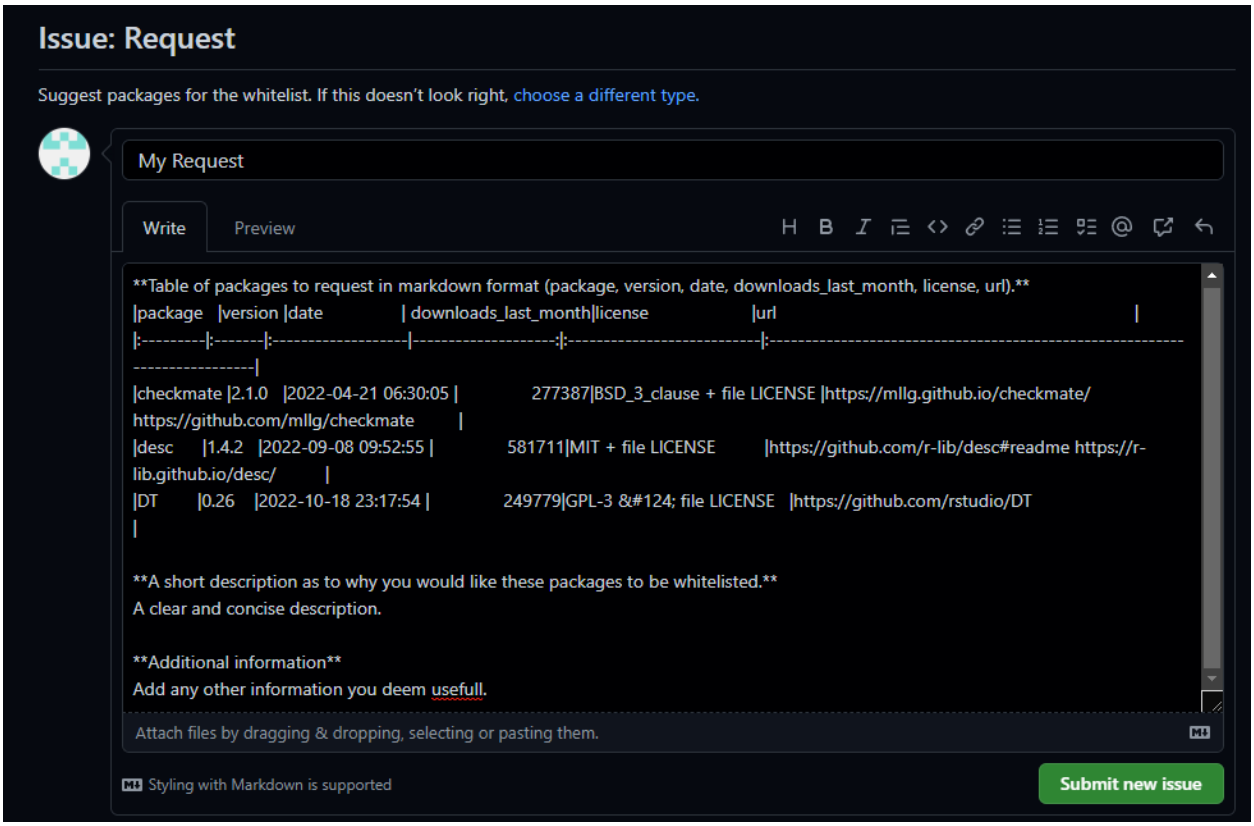


Figure 2: Request template

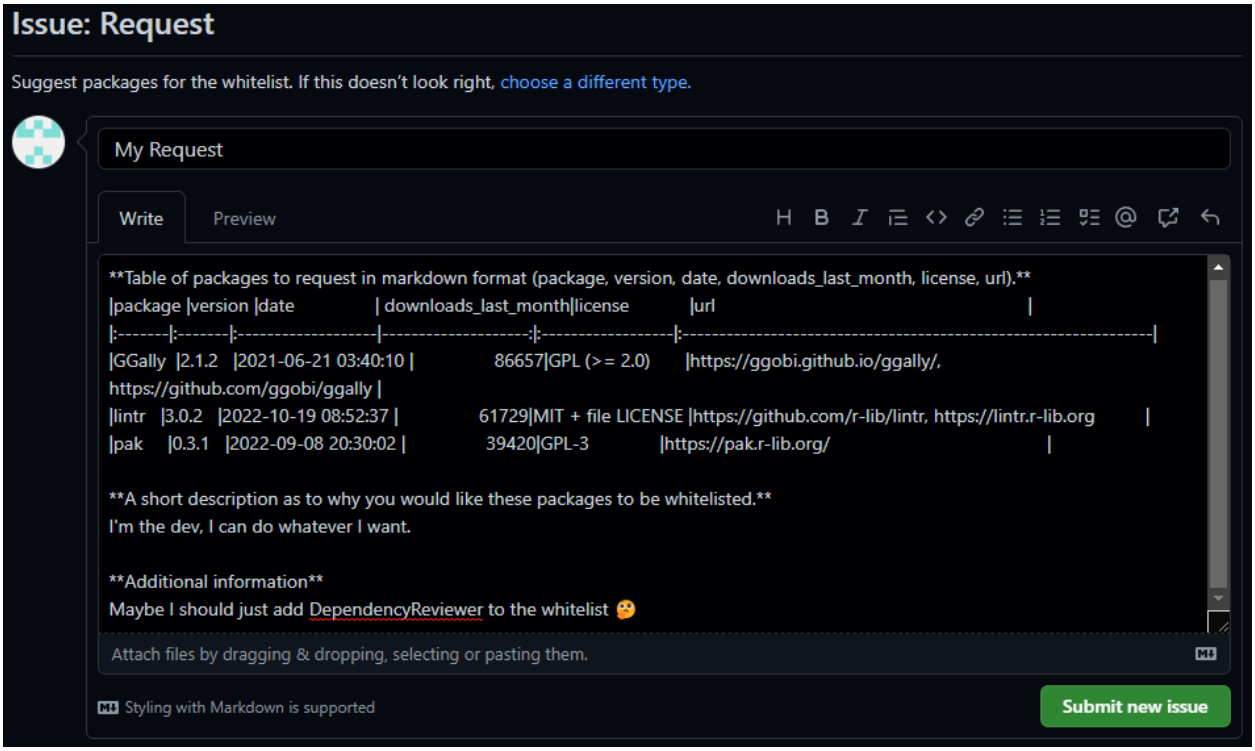
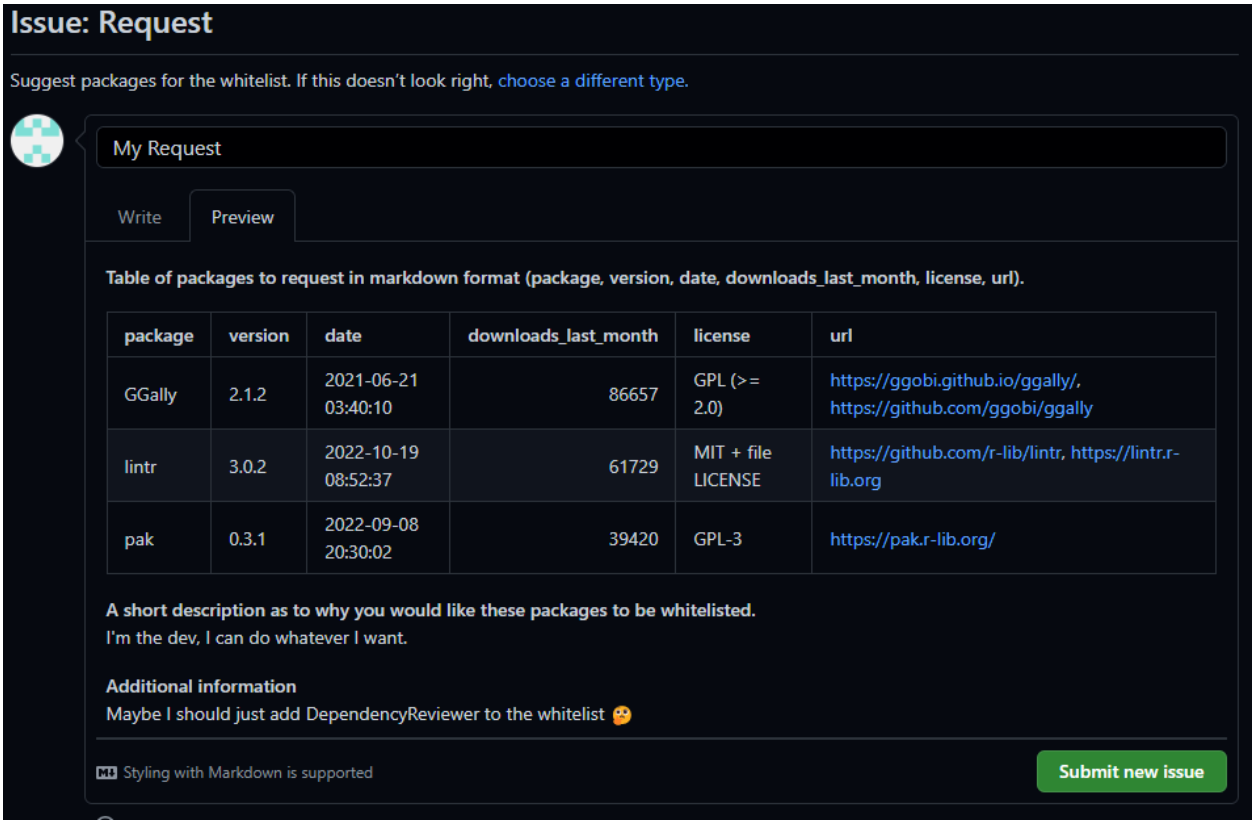


Figure 3: Request filled out



Issue: Request

Suggest packages for the whitelist. If this doesn't look right, choose a different type.

My Request

Write | Preview

Table of packages to request in markdown format (package, version, date, downloads_last_month, license, url).

package	version	date	downloads_last_month	license	url
GGally	2.1.2	2021-06-21 03:40:10	86657	GPL (>= 2.0)	https://ggobi.github.io/ggally/, https://github.com/ggobi/ggally
lintr	3.0.2	2022-10-19 08:52:37	61729	MIT + file LICENSE	https://github.com/r-lib/lintr, https://lintr.r-lib.org
pak	0.3.1	2022-09-08 20:30:02	39420	GPL-3	https://pak.r-lib.org/

A short description as to why you would like these packages to be whitelisted.
I'm the dev, I can do whatever I want.

Additional information
Maybe I should just add DependencyReviewer to the whitelist 🤔

Styling with Markdown is supported

Submit new issue

Figure 4: Request preview

What does it need? `summariseFunctionUse` has several optional arguments:

1. **r_files** default (list.files(here::here("R"))): If `in_package = TRUE` expects a character vector of at least length 1 of file names in the */R/* folder. If `in_package == FALSE` expects full paths to the R-files.
2. **verbose** default (FALSE): If `verbose = TRUE` will print messages in the console on which file the function is currently working. Useful when reviewing large R-files. If `verbose = FALSE` will not print said messages.
3. **in_package** default (TRUE): If `in_package = TRUE` will expect that the function is run inside a package-project. If `in_package = FALSE` will expect that the function is run outside a package-project and will expect full file paths to the files reviewed.

By default `summariseFunctionUse` will expect that it is ran inside a package-project and will look at the */R/* folder inside the project.

What does it return? `summariseFunctionUse` returns a class of data.frame with the following columns: *r_file*, *line*, *pkg*, *fun*.

```
# Assumes the function is run inside a package-project.
datatable(
  summariseFunctionUse(list.files(here::here("R"), full.names = TRUE)
))
```

Show 10 entries Search:

	r_file	line	pkg	fun
1	checkDependencies.R	27	base	function
2	checkDependencies.R	29	dplyr	filter
3	checkDependencies.R	29	base	is.na
4	checkDependencies.R	30	dplyr	rename
5	checkDependencies.R	31	dplyr	left_join
6	checkDependencies.R	33	base	c
7	checkDependencies.R	35	dplyr	filter
8	checkDependencies.R	48	base	function
9	checkDependencies.R	51	dplyr	filter
10	checkDependencies.R	52	dplyr	anti_join

Showing 1 to 10 of 243 entries

Previous 1 2 3 4 5 … 25 Next

```r
if (interactive()) {
  # Any other R-file, with verbose messages
  foundFuns <- summariseFunctionUse(
    r_files = "../inst/testScript.R",
    verbose = TRUE
  )

  datatable(foundFuns)
}
```

The found functions can then be plotted out for each package. For the sake of this demonstration, only a few packages will be plotted.

```r
if (interactive()) {
  funCounts <- foundFuns %>%
    group_by(fun, pkg, name = "n") %>%
    tally() %>%
    dplyr::filter(pkg %in% c("checkmate", "DBI", "dplyr"))

  ggplot(
    data = funCounts,
    mapping = aes(x = fun, y = n, fill = pkg)
  ) +
    geom_col() +
    facet_wrap(
      vars(pkg),
      scales = "free_x",
      ncol = 1
    ) +
    theme_bw() +
    theme(
      legend.position = "none",
      axis.text.x = (element_text(angle = 45, hjust = 1, vjust = 1))
    )
}
```

getGraphData

What does it do? `getGraphData` creates an igraph graph object of all the dependencies that the root package depends on. This includes direct and transitive dependencies.

What does it need? `getGraphData` has three optional parameters:

1. **path** default (here::here()): Path to the package to get the graph data of. By default assumes that the function is ran inside a package-project.
2. **excluded_packages** default (c(" ")): A character vector of packages to be excluded. By default is empty.
3. **package_types** default (c("imports", "depends")): Package dependency types to be included. By default imports and depends are included. Available types are: **1**) *"imports"*, **2**) *"depends"*, **3**) *"suggests"*, **4**) *"enhances"*, **5**) *"linkingto"*

Without any of these specified, the `getGraphData` function assumes that it is ran inside an package-project.

What does it return? `getGraphData` returns a class of igraph.

```
graphData <- getGraphData()
```

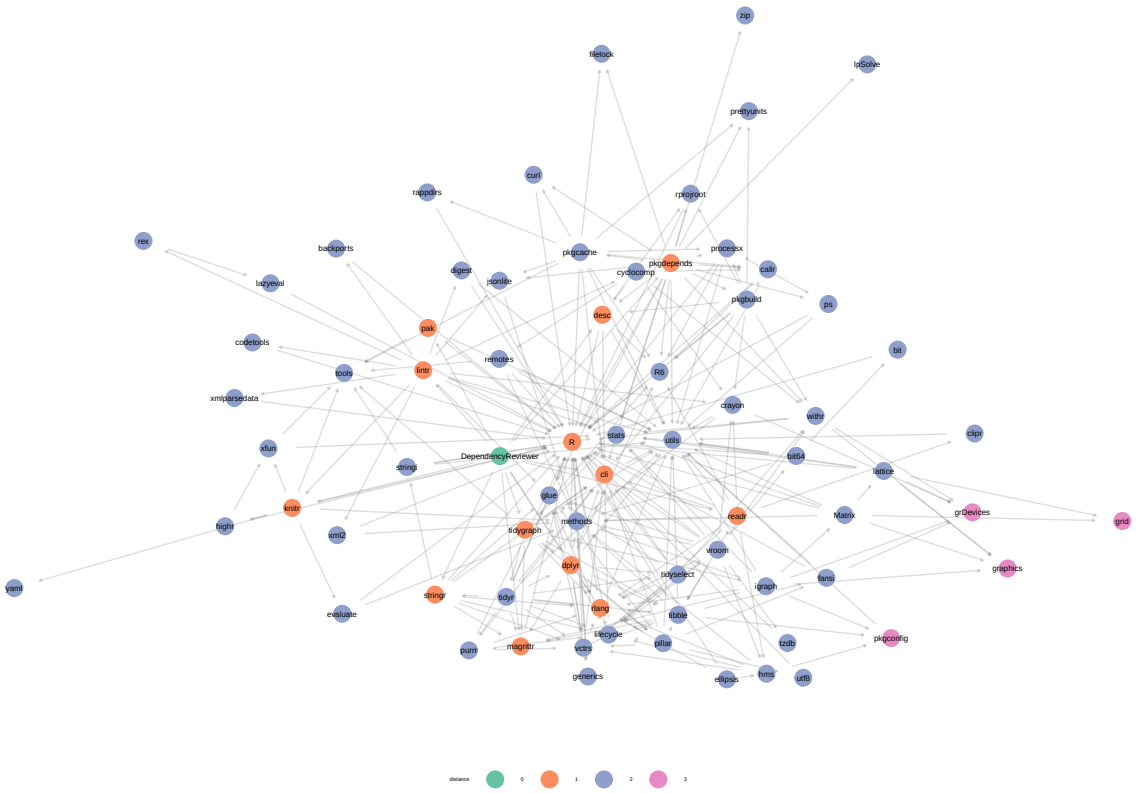
Because the amount of dependencies in the graph quickly get out of hand, it is suggested that you would either filter the igraph object after the fact, or only look at one kind of package type. In the following example we'll look at *"Imports"* only to keeps things simple.

It could then be plotted like so:

```
# Get graphData with only imports
graphData <- DependencyReviewer::getGraphData()
```

```
# Calculate colour of nodes based on distances from root package
cols <- factor(as.character(apply(
  X = distances(graphData, V(graphData)[1]),
  MARGIN = 2,
  FUN = max
)))

# Plot graph
ggnet2(
  net = graphData,
  arrow.size = 1,
  arrow.gap = 0.025,
  label = TRUE,
  palette = "Set2",
  color.legend = "distance",
  color = cols,
  legend.position = "bottom",
  edge.alpha = 0.25,
  node.size = 2.5,
  label.size = 1,
  legend.size = 2
)
```



runShiny

What does it do? `runShiny` runs a local shiny app that houses all the before mentioned functionality in one environment. `runSHiny` assumes that it is being ran inside a package-project.

What does it need? `runShiny` Takes no arguments

What does it return? runShiny returns a class of shiny.appobj.

```
runShiny()
```

The shiny application has three main tabs: **1**) Package review, **2**) Dependency Graph, and **3**) Path to dependency.

Package review

On the package review tab there are three main panels.

1. **Settings**: The settings have two parts on this panel: A file picker, and tick boxes to packages. Currently all the files are in the summariseFunctionUse table.
2. **summariseFunctionUse table and plot**: The summariseFunctionUse table for the specified files, or all files if *ALL* is picked in the file picker in the settings.
3. **Script preview**: A preview of the contents of the selected file. If *ALL* is chosen, a dummy script will appear, or the last viewed contents will stay.

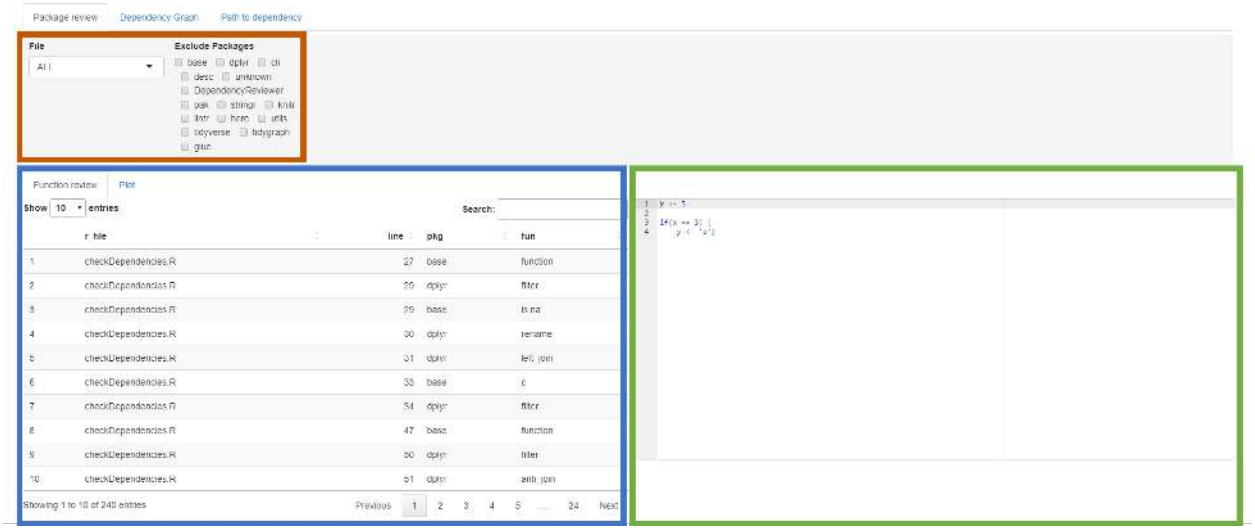


Figure 5: Function review

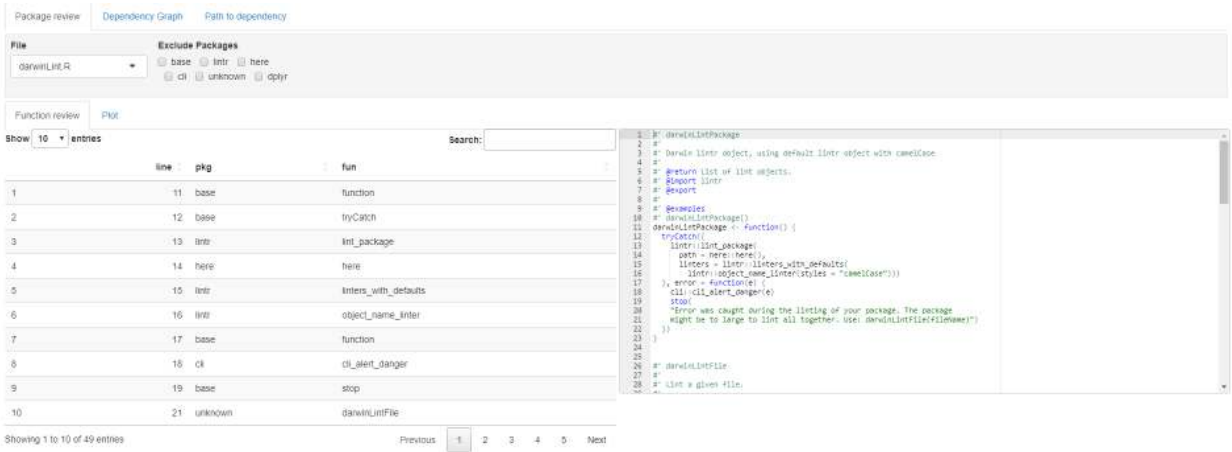


Figure 6: Package review

Notice how the **Settings**, **summariseFunctionUse table and plot**, and **Script preview** dynamically change when the darwinLint.R file is selected.

When swapping from the **Function review** to the **Plot** tab a bar graph is shown for each package used in the file. The bars represent the amount of function calls in that file per package.

Figure 7: Function review plot

Lets say base functions are not interesting for your use case, you can then tick the *base* tick box in the Exclude Packages in the settings.

base packages are now excluded from both the **summariseFunctionUse table and plot**.

Dependency Graph

The Dependency Graph tab displays a graph, like plotted earlier, using the `graphData` function. On the right-hand-side different kinds of dependencies are able to be chosen to be included in the graph.

Path to dependency

The path to dependency tab displays how the root package depends on any recursive dependency.

On the right-hand-side a dependency found somewhere included in the root package can be chosen. A cutoff can be defined to limit the distance from the root package to the chosen dependency.

darwinLintFile

What does it do? `darwinLintFile` is an extension of the default Lintr object, but instead of *snake_case*, it uses *camelCase*. As the name suggest it will run the lintr on a specified file.

What does it need? `darwinLintFile` takes one parameter: 1. **fileName**: Path to an R-file.

What does it return? It returns a class of lints.

However the output of a lintr function can be cast to a data.frame.

Figure 8: Package review

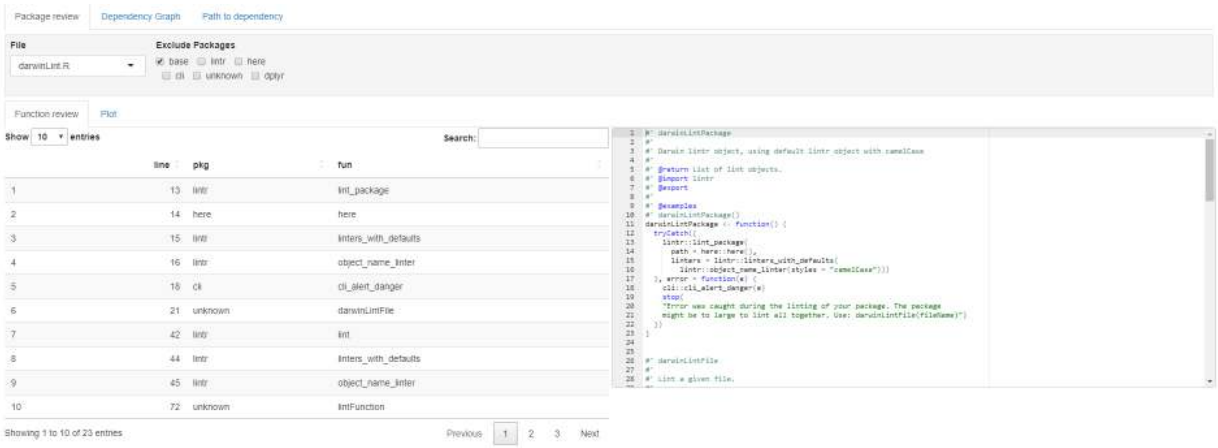


Figure 9: Package review

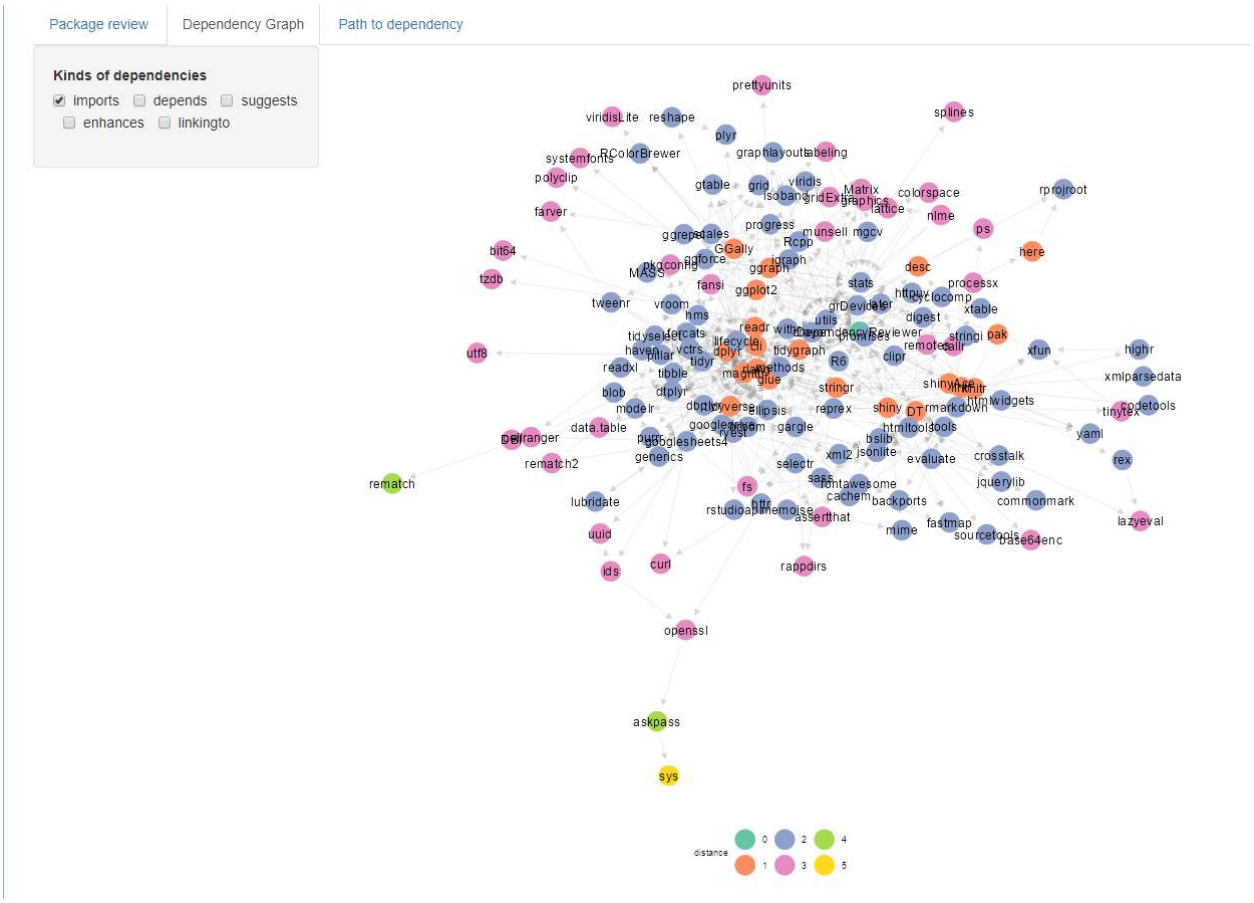


Figure 10: Package review

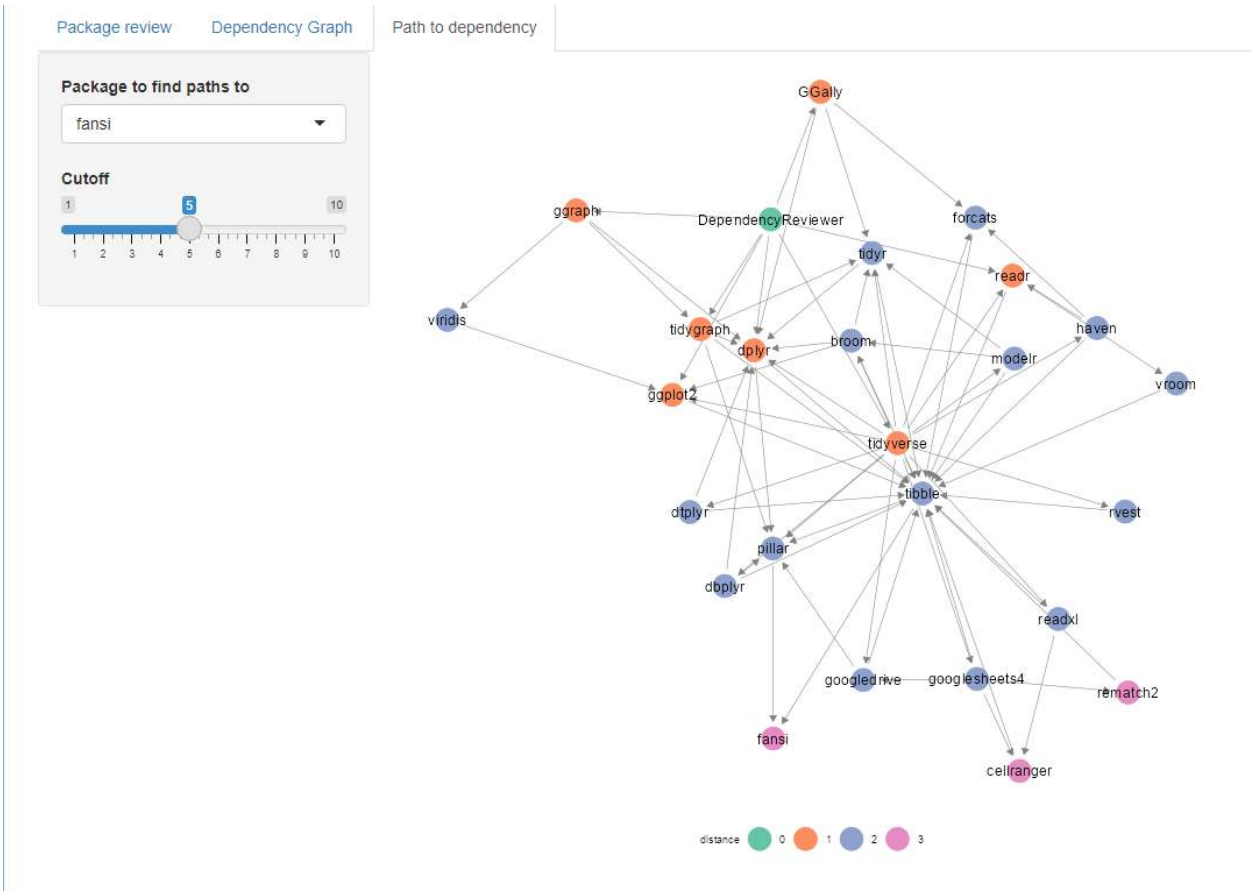


Figure 11: Package review

```r
if (interactive()) {
  lintOut <- data.frame(
    darwinLintFile(
      fileName = "../inst/testScript.R"
    )
  )
}
```

Which can then be manipulated to get a summary of lint messages.

```r
if (interactive()) {
  lintOut %>%
    group_by(type, message) %>%
    tally(sort = TRUE) %>%
    datatable()
}
```

darwinLintPackage

What does it do? `darwinLintPackage` is an extension of the default Lintr object, but instead of *snake_case*, it uses *camelCase*. But unlike `darwinLintFile`, will run the lintr on the entire package. Therefore it will assume that the function is ran inside a package-project.

What does it need? `darwinLintPackage` Does not take any arguments.

What does it return? It returns a class of lints.

darwinLintScore

What does it do? `darwinLintScore` calculates a percentage per type of lint-message from the lintr.

The percentage is calculated as:

$$darwinLintScore_{type} = \frac{n_{messages}}{n_{lines}} \times 100$$

What does it need? `darwinLintScore` takes one predefined argument: 1. **lintFunction**: A lint function extended from `lintr::lint_package` or `lintr::lint` 2. **...**: Any other arguments that the lint function might need

What does it return? Returns a class of data.frame with two columns: **1**) type, and **2**) pct.

It will also print out colour coded messages with the percentages per message type.

```r
if (interactive()) {
  darwinLintScore(darwinLintPackage)
}
```

```
i style: 5.9% of lines of code have linting messages
i warning: 0.95% of lines of code have linting messages
```

type	pct
style	5.9
warning	0.95