Creating a Shiny App to Display Linting Information

Problem: Outputs of the Linter package cluttering Slack.

Goal: Create a Shiny web app to centralize linting data.

During the second half of my time here, I worked on expanding my Linter package. As different entities and fields were added in Benchling, I wrote the appropriate functions and scripts to lint them. I also used Excel to reformat and upload this new data to Benchling. As part of this expansion process, I consolidated my existing code to better adhere to the tenets of functional programming, eliminating repetitive code. I learned how to use roxygen to document my package and did my best to adhere to the standards in place. I also worked on linting the AWS S3 LodoAmpliconsProcessed bucket. This linting went beyond checking for properly formatted names and required cross-referencing different files and their contents to ensure that no data was missing. Additionally, I checked that sequence runs in Benchling had corresponding S3 buckets. The largest obstacle when doing S3 linting was the sheer volume of data that had to be recursively examined. Testing my code with the actual dataset could take upward of half an hour each time, and it could be very frustrating to run all that only to discover a typo.

Enter the LodoTest bucket. Zdk pointed me to this bucket and gave me write credentials. I created a barebones mirror of the LodoAmpliconsProcessed bucket to use in package tests. (This had to be done partially through the command line due to a strange glitch that caused S3 to make an invisible empty file upon creation of a folder using the “New Folder” button.) This could also be used to speed up the process when I was working on other things that required time-consuming calls to S3. I redesigned all the tests in the Linter package to draw from preexisting datasets, either files in the package or using the LodoTest bucket.

Another part of the project I worked on was summarizing the connections between different Benchling entities. I did this with a series of functions collected in the file Aggregation.R. For these functions, I learned how to merge and manipulate multiple dataframes. I also used the ggmap package, which utilizes Google Maps, to identify the origin state of soils based on their latitude and longitude. A major issue I ran into doing this was that the Google Maps server frequently did not respond, leading to a “bad coordinates” warning message. To reliably get a location, I had to ping the server multiple times. This caused the function to be quite slow, and zdk later rewrote it using functionality he had already created which did not use Google Maps, did not require internet connection, and markedly sped up the process.

As a short aside, I helped update .yml files to connect with Drone. I updated the file in the Regex package to point to the correct locations and to run at different times.

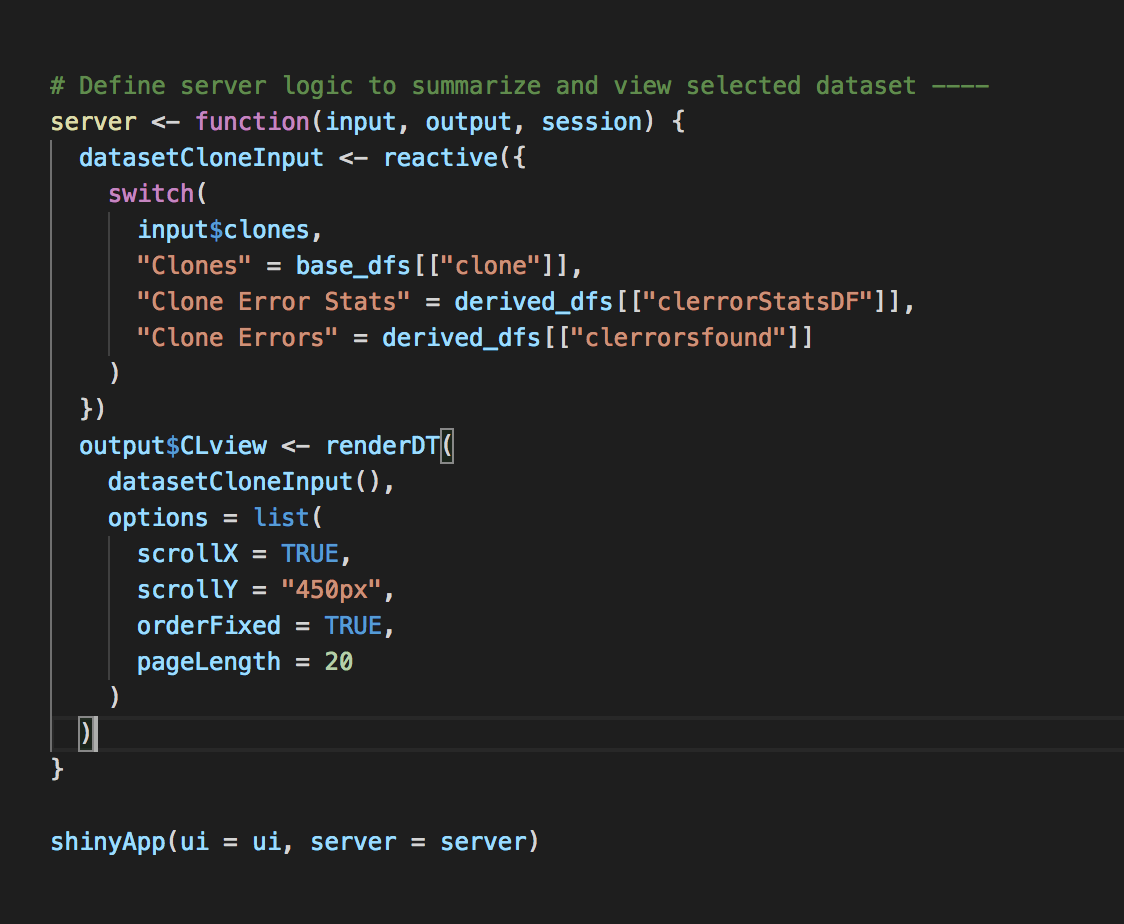
The primary product of the last month or two is the creation of my datalinter Shiny app. I first began experimenting with Shiny on Zcp’s suggestion, following online tutorials. The first iteration of the app just had a dropdown list that displayed the datatables for error summaries and the original Benchling data. I went on to rework some of my Linter package code so that all the errors could be displayed, rather than just summaries. I added different tabs for neatness and clarity’s sake. The “Other Data” tab differs slightly from the others in that it displays not the linting errors but the results of my Aggregation.R functions, but on the Shiny side it is roughly the same. Zcp requested I add a summary page, so I consolidated the error data into a couple highly visual bar graphs, color coded by volume of issues. I went through a couple different designs, finding that Shiny did not support my initial design, eventually landing on a two-graph solution that provided a visual of which entities had errors and how urgent these errors are. I then made an even shorter metastatistics summary, coded directly into the app via HTML because its design was too simplistic to be easily rendered with any of R’s built-in plots.

My top takeaways from this project: Always fetch/rebase from upstream before making a PR. Don’t use load\_all when working on a package that others will be using.

Things I Learned:

* Shiny
  + Basic app layout

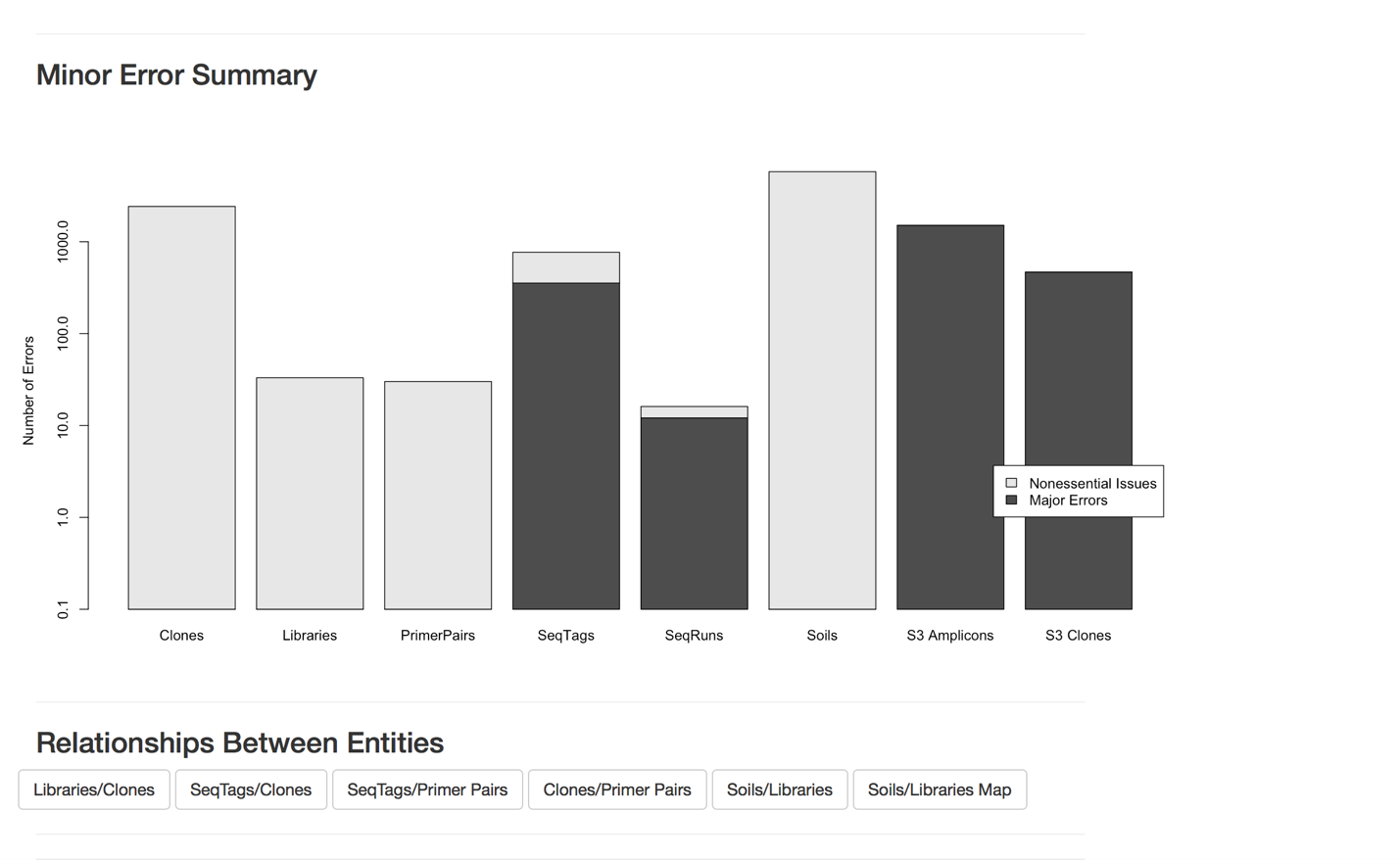
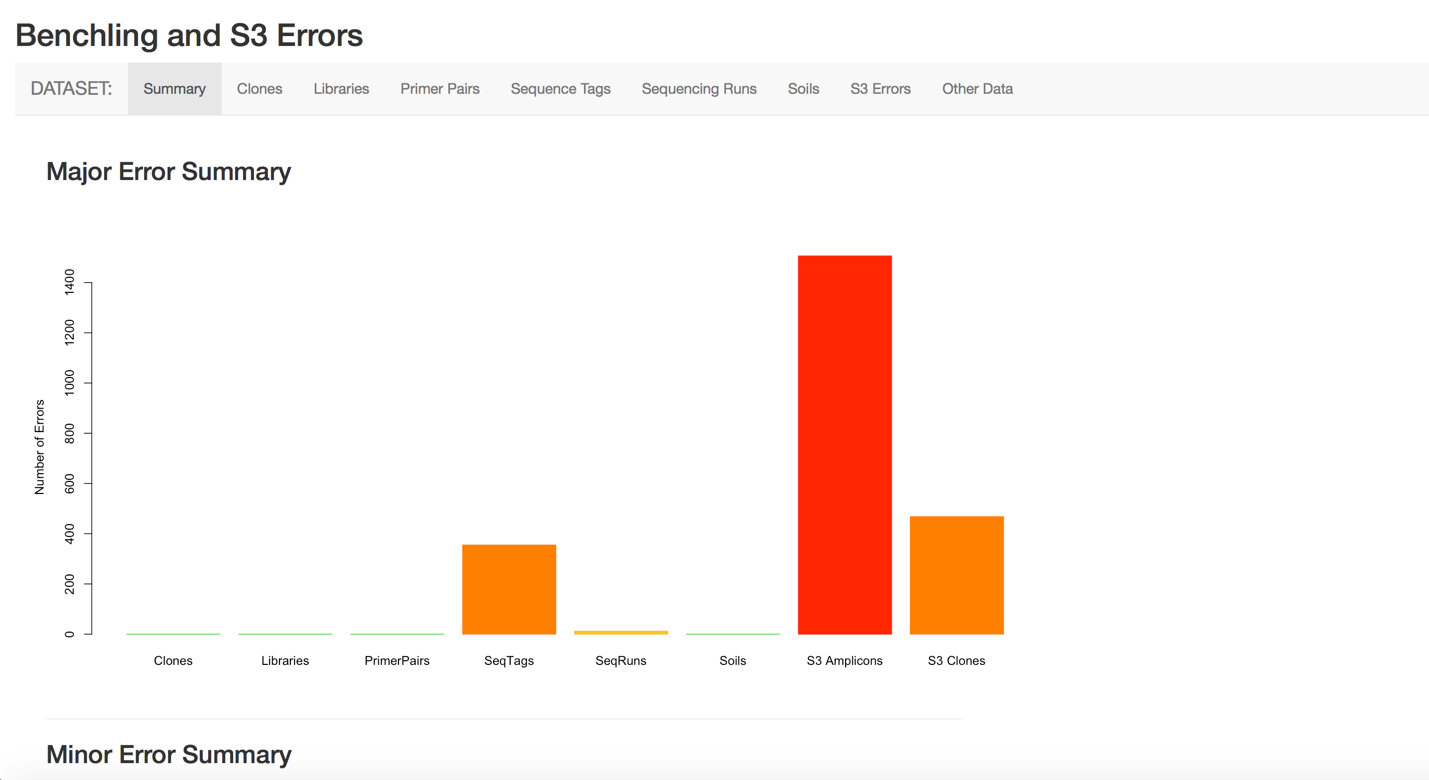




A Shiny app consists of a UI function and a server function. In the code above, the output shown in the server depends on named input in the UI. For instance, the input dropdown “clones” has its place in the UI and a reactive in the server dictating what information to display. Similarly, “CLview” is placed in the UI and rendered to show a datatable in the server. Shiny has a lot of built-in layout features, as seen in the many different panels above (e.g. “tabPanel”, “mainPanel”).

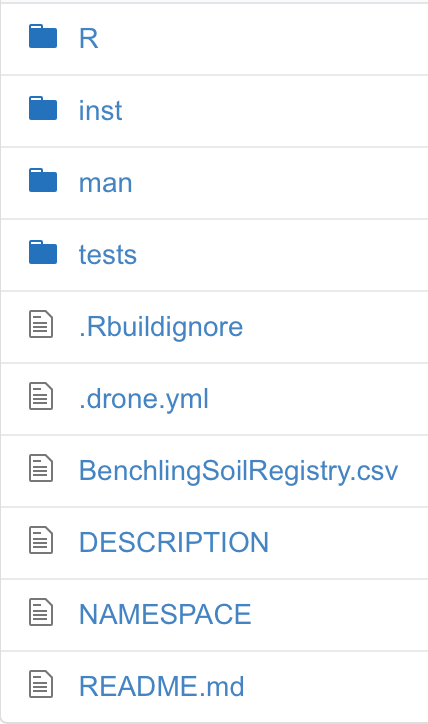
* + Plots

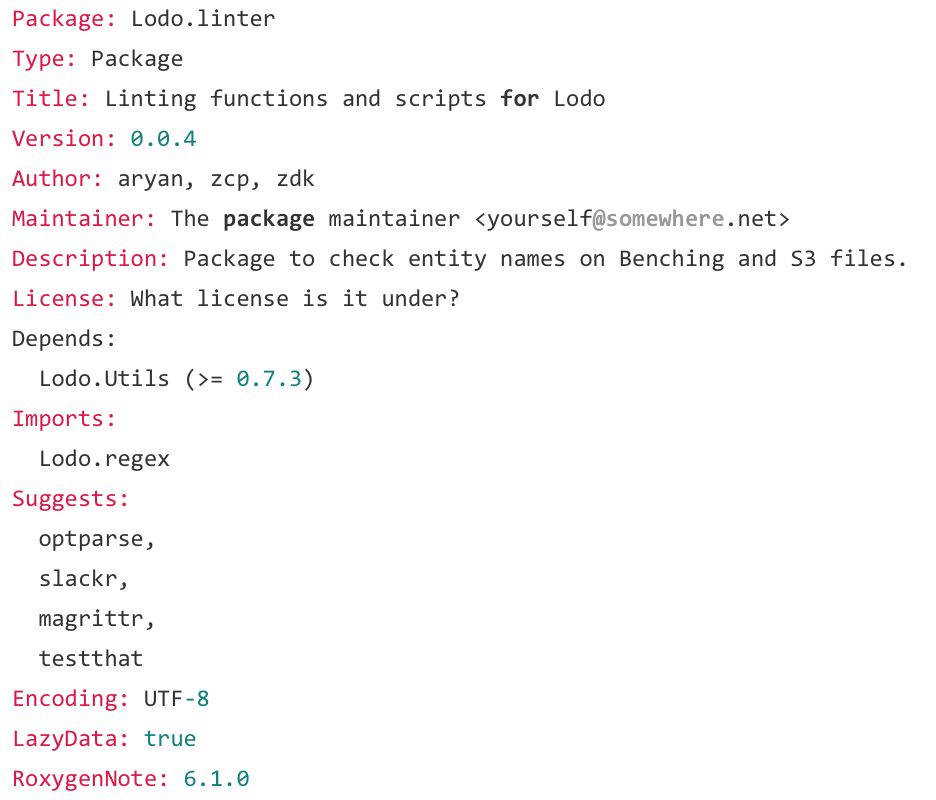
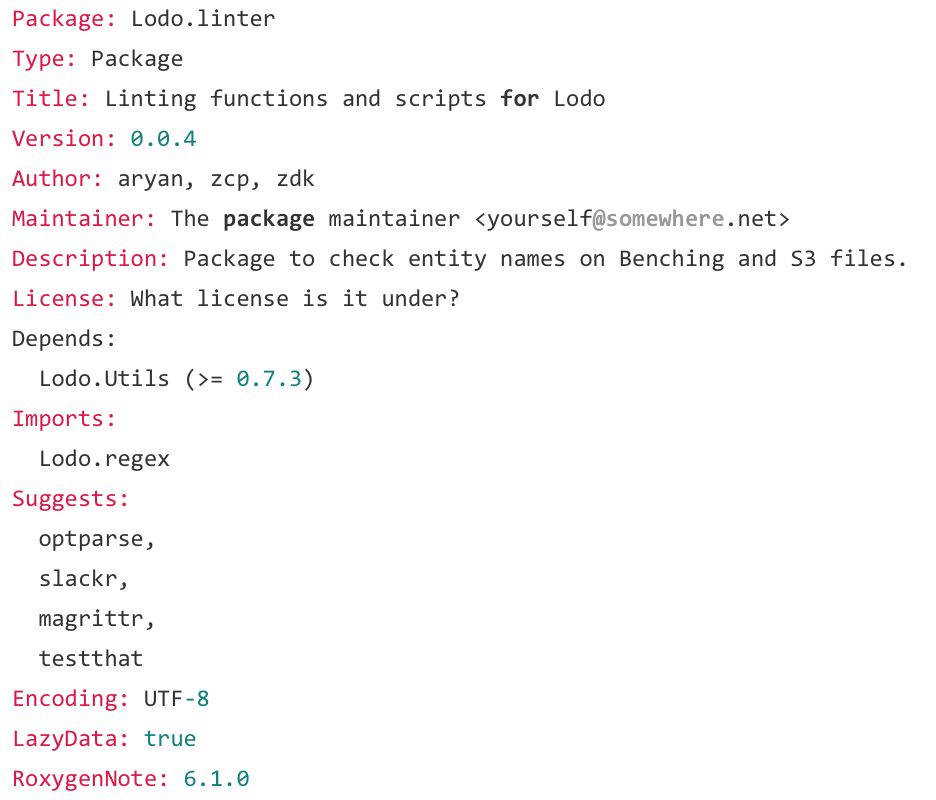
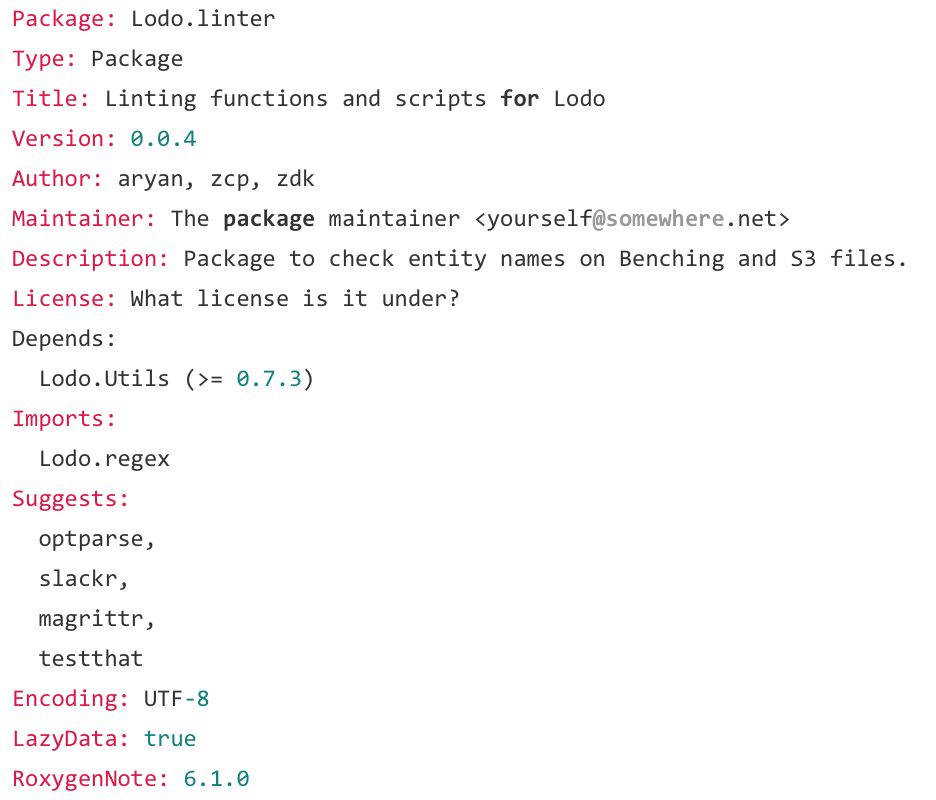




The functions above produce the respective bar plots below them. Plots are made pretty simple in Shiny, and I experimented with a few different things before settling on this. Different options under the renderPlot family of functions allow for a multitude of adjustments to the output, from scale to legend to colors.

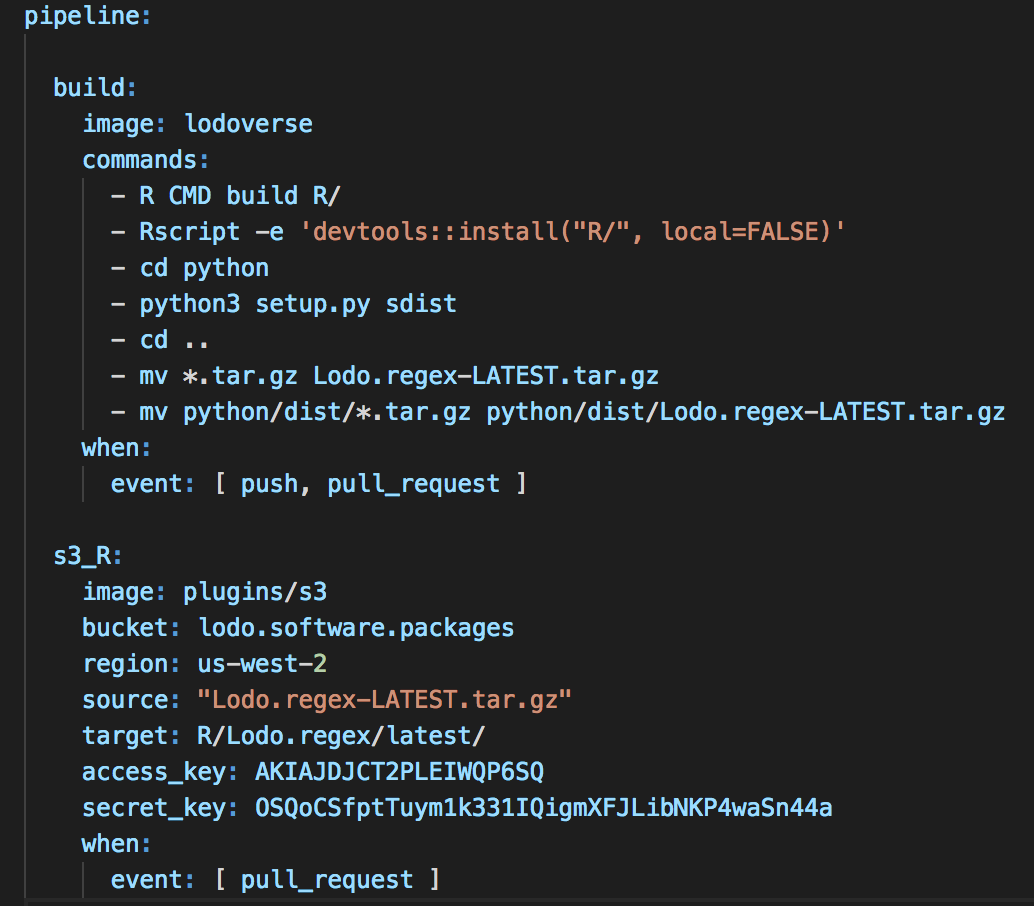
* R
  + Packages





At left are the directories and files that make up my package. It has a DESCRIPTION file, shown at right, detailing package, version, dependencies, etc. It has folders for the different portions of code (e.g. tests for all tests). It has a NAMESPACE file dictating package outputs and inputs. It has a README file explaining purpose, installation, and usage. Much of this was generated with roxygen.

* YAML



This is part of a file that syncs to drone.io, an automation platform for pipelines. Written in a list format, it specifies commands to execute on certain occasions (in these instances, linked to a git push/pull request) and where to find information.