Final Project Paper

Nathan McIntosh

Due December 14, 2020

1 Abstract

The Homebrew Package Manager is an open source command line tool for managing packages and applications for both macOS and Linux. This paper examines what data can be collected from Homebrew, how said data can be compiled, how it can be visualized, and what questions those might answer.

A sample of some of the questions one may be able to answer with the final visualization are:

- What happens if this packages suddenly stops working for everyone?
- Which packages have seen the most growth in the last year?
- Which packages are most popular and are staying most popular?

These questions are rather general and could be answered in a number of ways. This paper will examine the process of building up products that can answer these questions.

2 Introduction

A tool that I have come to love is the Homebrew package manager for Mac OS and Linux. Often I find myself wanting to test out a new software tool, or make sure that a setup "just works". It allows for easy installation of a tool and all its dependencies, easy uninstalling of tools no longer needed, and easy upgrading and downgrading of versions. As I watch it install, update, remove, etc.. packages, I'm always interested to see what dependencies are required, upgraded, downgraded. With the command `brew info <package name>` one can see statistics about the package. On my mac, Microsoft's .NET platform has the dependencies cmake, pkg-config, curl, icu4c, openssl, and Xcode. At the time of writing, it has the following statistics:

- install: 1,547 (30 days), 3,052 (90 days), 3,095 (365 days)
- install-on-request: 1,196 (30 days), 2,283 (90 days), 2,326 (365 days)
- build-error: 0 (30 days)

From this we can clearly see that either .NET is not yet used on a large scale on Mac OS, or more likely, Homebrew is not the primary means by which it is installed. What could we learn if we looked at this information for all the most popular packages Homebrew serves on both Mac and Linux? What I hope to accomplish in my final project is an overview of the state of Homebrew dependencies on Mac and Linux. Which are asked for most often? Which are downloaded as dependencies most often? How are popular packages interconnected? Which versions of macOS are being used the most?

3 Background

I was initially delighted to discover the analytics portion of Homebrew's site; it offers a lot of insight. However, it only offers as much insight as one can glean from reading number from a table. In other words, while helpful for getting specific numbers, a table doesn't allow humans to intuitively and accurately understand scale, change, relative sizes. Furthermore, it is even harder to understand the scale and organization of connected entities without a visualization.

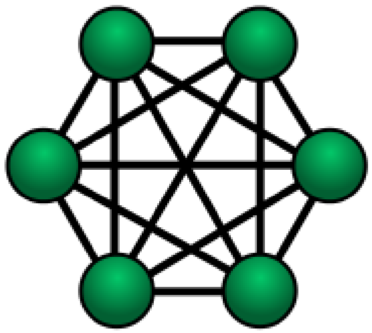


Figure 1: connected scatter plot

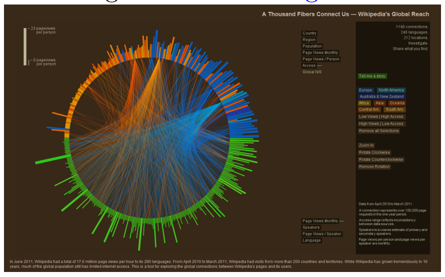


Figure 2: chord diagram

Visualizing dependencies can be done numerous ways, for numerous purposes. One may want to get a general sense for an ecosystem of connected entities, or see exactly which tasks must be accomplished to continue onto the next. One might want to see where time might be most well spent in shoring up the security of a tool. For example, the OpenSSL library (which is immensely popular as we will later see) has had a host of notable vulnerabilities, and as of May 2019, only two full time employees. One may wish to use packages with the fewest dependencies to reduce chances of dependency

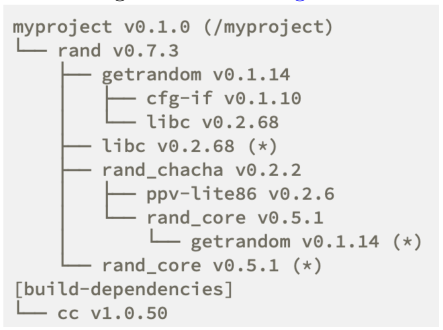
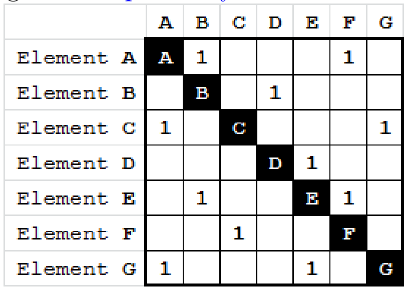


Figure 3: tree diagram

```
myproject v0.1.0 (/myproject)
└── rand v0.7.3
    ├── getrandom v0.1.14
    │   ├── cfg-if v0.1.10
    │   └── libc v0.2.68
    ├── libc v0.2.68 (*)
    ├── rand_chacha v0.2.2
    │   ├── ppv-lite86 v0.2.6
    │   └── rand_core v0.5.1
    │       └── getrandom v0.1.14 (*)
    └── rand_core v0.5.1 (*)
[build-dependencies]
└── cc v1.0.50
```

hell or a pad-left dependency breakage.

Figure 4: dependency structure matrix



	A	B	C	D	E	F	G
Element A	A	1				1	
Element B		B		1			
Element C	1		C				1
Element D				D	1		
Element E		1			E	1	
Element F			1			F	
Element G	1				1		G

In the case of viewing a system of connected entities, one can use connected scatter plots (fig. 1), or a chord diagram (fig. 2) of lines or filled-in ribbons. To see dependencies of a single item, one might want a simple tree structure (fig. 3) like that used by Cargo or the command line. Finally, one can also create a dependency structure matrix (also called a design structure matrix) to see a simple matrix/heat-map (fig. 4) of dependencies.

In this project, we will see two of these methods, and use them to answer various questions about the Homebrew ecosystem and package dependencies.

4 Approach

This project contained four main stages:

1. Collecting all of the relevant data
2. Joining said into usable data structures
3. Creating individual figures that each tell a story
4. Organizing said figures into a single visualization

All data was collected using Homebrew's analytics API, which produces JSON. Almost all of the links on the analytics homepage were queried, which produces data on total downloads, and downloads at human request (i.e. not as a dependency to another package). This data was then joined into a single large table of about 47k rows.

On top of that, most individual packages have their own info API. The information from almost all of these packages were gathered as well; a task that takes about 20 minutes using all 8 cores on my computer, and produced a combined 10.8 MB JSON file once combined. Querying this JSON allows for tables of how many packages depend on a given package, and tables where each row shows a single dependency. e.g.

$$A \to B$$

$$A \to C$$

$$C \to D$$

$$\vdots \to \vdots$$

It is this data of dependencies that was hardest to collect and put together appropriately. But it is also this data that we can use to produce some of the most interesting dependency visualizations.

The third and fourth stages are creating individual figures, and putting them together. The idea here is that each figure tells its own unique story, and together they allow the viewer to build up a more complete idea of the state of Homebrew. To create the individual plots, the Plotly library for Python was used. I chose it because I am familiar with it for work, and it produces cleanly styled, interactive visualizations. Plotly alone can be used to combine many plots in many subplots, and it can be used to add elements such as, buttons, sliders, and drop down menus. However, putting together the plots in the manner envisioned proved easier with Plotly's Dash library.

5 Results

The final visualization is a local web page hosted by a server running from one's own computer. It contains four visualization panels as seen in figure 5.

1. In the top left (fig. 6), a dependency structure matrix of the top 100 most depended upon packages. Since almost there are almost 7000 package dependencies (at the time of writing), I deemed it unrealistic to show all of these connections. However, if we down-select to just the top 100 most depended upon packages, we can potentially plot this data. A data point on the chart produced can be read as "package, at y location y_i depends on package at x location x_i." A column with many data points means that that package is used by many of the other packages in the top 100. A row with many data points means that that package depends upon many other packages in the top 100. This format was the easiest of the dependency graphs to create with Plotly. Indeed, connected scatter plots and chord diagrams proved very difficult.

2. In the bottom left of the webpage (fig. 7) is a line chart of package popularity. The user can choose between MacOS or Linux. The plot describes the popularity of the 20 most popular packages over the

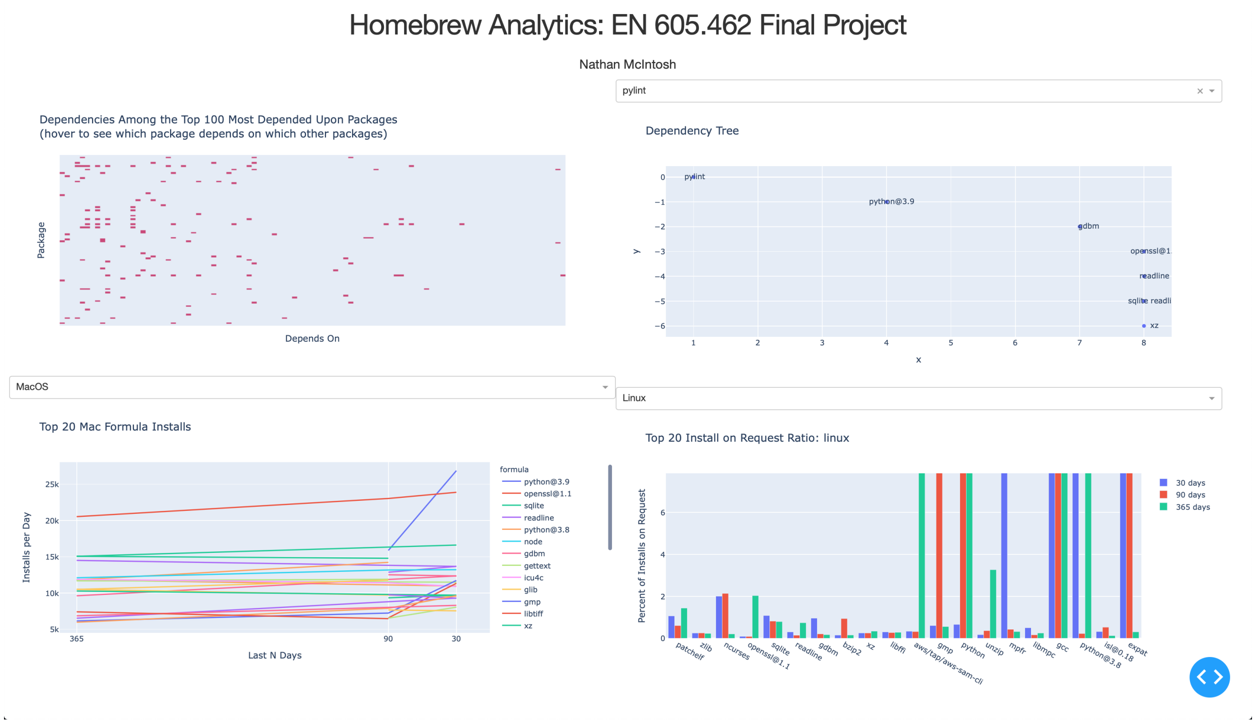


Figure 5: Summary View of the Final Visualization

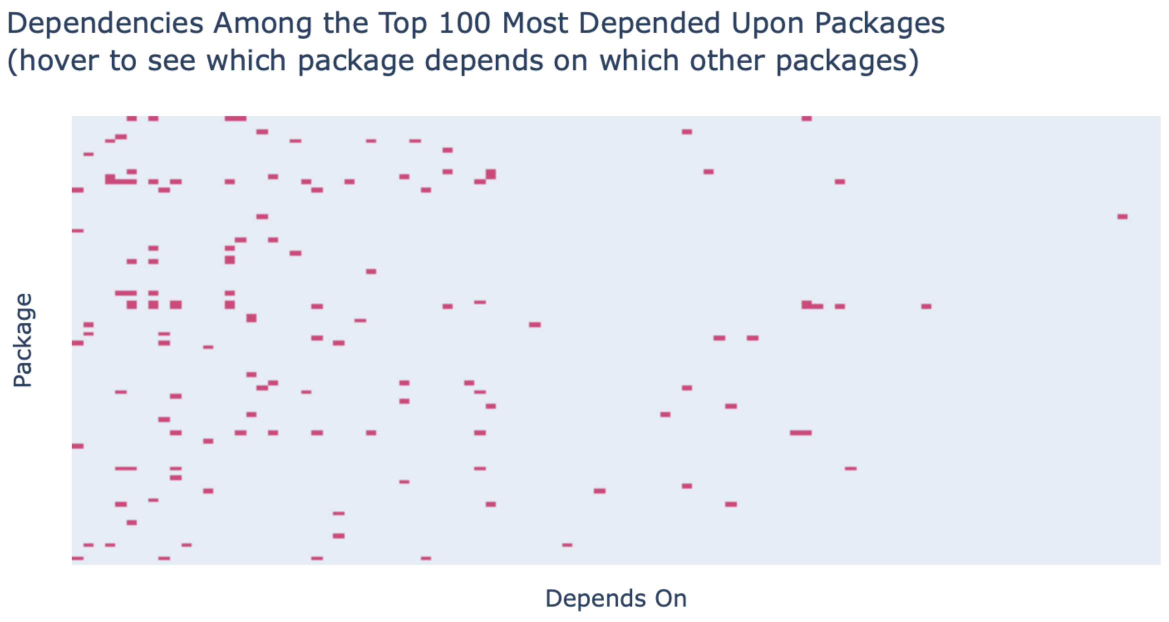


Figure 6: Dependency Structure Matrix

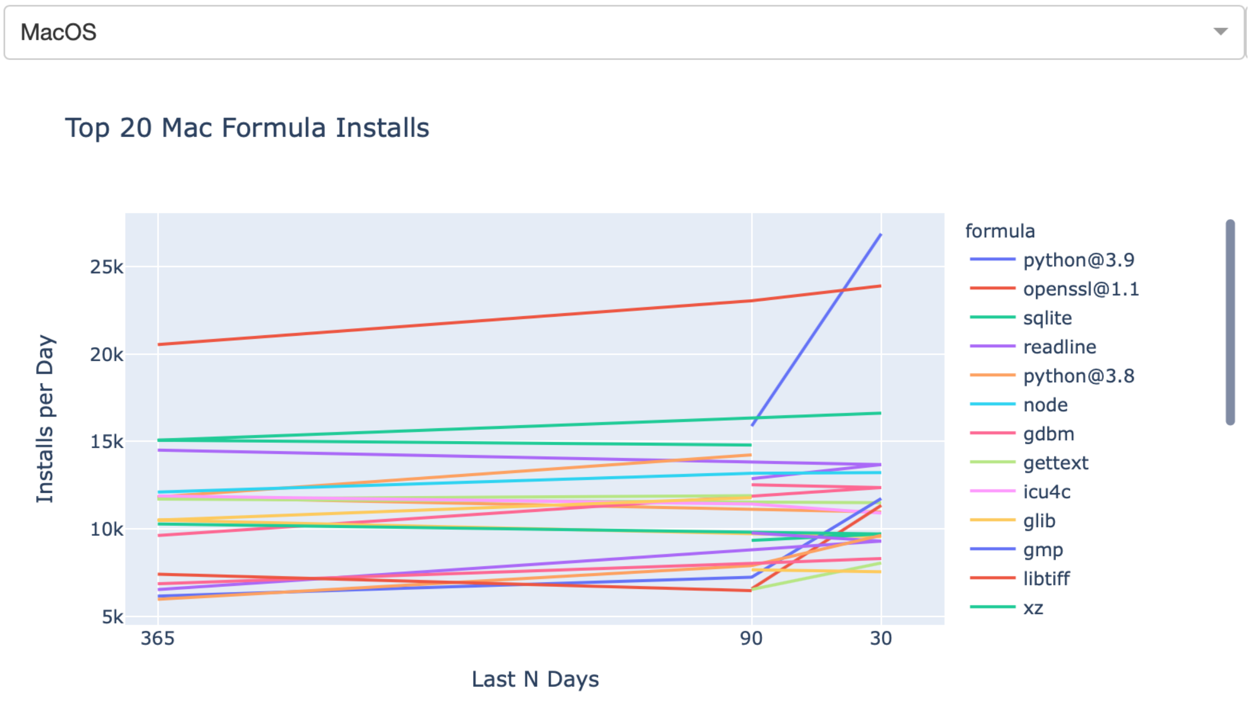


Figure 7: Package Popularity

the last 365, 90, and 30 days, in terms of a normalized scale. The scale was normalized so that the count of downloads in the last 365 days wouldn't greatly overshadow values in the last 30 days.

3. In the top right of the webpage, (fig. 8) is a plot attempting to mimic the `tree` command. At the moment, I have been unable to connect the nodes to each other with proper lines. It has a dropdown that allows the user to select any of the 2774 packages that have dependencies. Creating this plot has proven more difficult than expected, due to its tree like nature. To get around the complex recursion generally required to produce the x and y positions of a tree structure like this, I captured the output of python's pprint function, and gathered x and y data from the string it produced.

4. Finally, in the bottom right of the webpage, (fig. 9) is a grouped bar chart of the install on request ratio for the top 20 most popular packages for both operating systems. The y-axis is the percent of installations on (human) request. Something close to 100% means that it is mostly installed when humans request it with `brew install <package-name>`. Some notable packages here are `aws-sam-cli`, `gmp`, `gcc`, `unbound`, and `git`. In contrast, values close to 0% mean that a package was mostly installed as a dependency for some other package. On the Linux side there are some particularly notable packages such as `libffi`, `zlib`, etc.

As well as interesting contrasts between different packages, there are notably odd patterns for individual

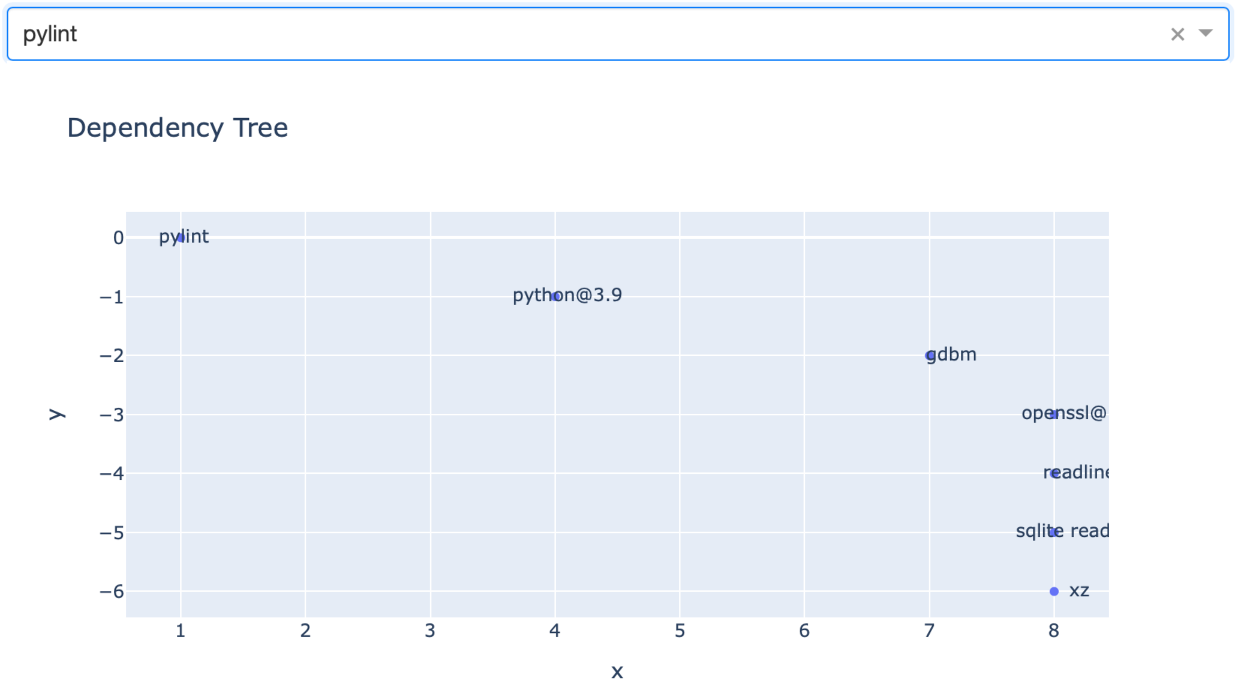


Figure 8: Single Package Dependency

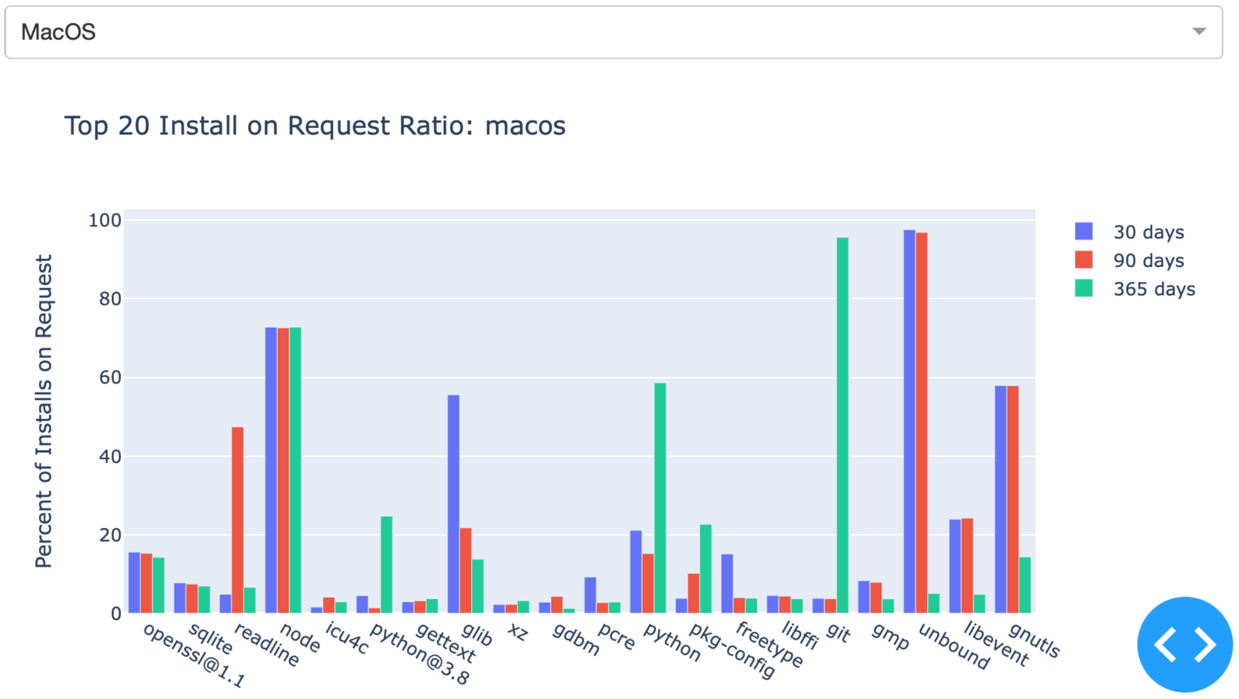


Figure 9: Install on Request Ratio

packages across different time spans. In fig. 9 for instance, we can see that git was highly requested over the last 365 days, but almost not at all in the last 90, or 30 days. What could cause such odd behavior? Is there a specific time of year when people generally install git?

It should be noted that I attempted to use Graphviz for displaying the dependencies between all packages. However, the resulting graph was so convoluted, that I decided not to include it in the main visualization.

6 Conclusion

This project effectively collects, cleans, and visualizes statistics of Homebrew. I wrote an effective set of tools for collecting information from Homebrew via its analytics API. This involved requesting JSON; parsing the JSON, and marshalling the JSON into tabular format. I created multiple, unique visualizations, answering different questions, for multiple time spans and both operating systems. Finally, all of the visualizations were combined in a single web page, generated using Plotly's Dash "framework for building web analytic applications". The entire pipeline for collecting, cleaning, and visualizing the data can be run from a single Python script.