Container Explosion, Slumlord, Caching, Binder

Kevin
School of Electrical and Computer Engineering

Tim and Doug
Twentieth Century Fox

Kyle and Ian
Starfleet Academy

***Abstract*—Potential target: https://www.canopie-hpc.org/cfp21/**

I. Introduction

II. Describe data

Data Collection

We used a publicly available dataset containing binder launch records in the form of two tables. The launches table contains 14 million launch records, with columns that show the timestamp of the launch and index to link to an image in the specifications table. In this second table, a recipe is given for every unique launch, with raw text specifying the required packages and optionally, their versions, when needed to build a Binder container.

Beyond the data presented in the launches and specifications datasets, there were a considerable amount of additional factors to explore when expanding the capabilities of our model, namely the size, installation time,list of dependencies, version history and various statistics for a given package. The most central of these would be the construction of a dependency tree, such that we are able to know the full dependencies for any Python package in pip. The importance of having a quick lookup to the full dependencies for a package extend to the ability to find the individual size and installation times for a package, and fixing the incomplete generation of the dependency table from the specification dataset. The size and installation times for a package were determined for each package through benchmarking and collecting sizes in a virtual environment. Finally, the version history and statistics such as Github stars or forks were pulled from pypi and libraries.io.

In order to determine the sizes, times and dependencies needed to install each Python package, we set up a new virtual environment for every package with only Python 3.6 and pip installed. After activating the environment, the package was installed using pip, with a function to benchmark the process. Among many other outputs, the size of the package and its list of dependencies could be determined through a command line argument. After a package was finished, the virtual environment would be destroyed and cache for the environment stored in the main instance would be removed, thus setting the stage for the next package and virtual environment. This process was repeated using a bash script, and every package was timed ten times to account for variation in installation times and connectivity to the network or instance. The benchmarking of 1000 packages took roughly 5 hours to complete.

A glaring issue that must be addressed is finding the *individual* installation time and size for a package. When a package is installed in an empty environment, it installs all of its dependencies as well, and this is factored in both the installation time and the size when the data is collected. Although this could be left as is for the time being, the problem would have to be solved somewhere down the line, so some strategies were used to utilize the dependencies list in resolving these issues at this stage. The first idea would be to run an algorithm that would, given the total size, time, and list of dependencies for each package, be able to separate them so that an individual size and time would be calculated for each package. The algorithm would repeatedly run iterations until a natural stopping point, where on each iteration, it would pop the packages with empty dependency lists, take their sizes and times, and ”subtract” them from all other packages containing them, while removing all instances of that package in the process. This method proved to be successful for sorting out sizes, but as it proved to be flawed for finding times (with an abundance of negative values at the end), a different strategy was applied directly in the bash script. Hinging on the fact that only the package itself (and not its dependencies) are uninstalled when attempting to uninstall the package, an effective add-on would simply be to uninstall the package and then benchmark the second installation of the package (which will only install the package, and not its dependencies).

A myriad of statistics can be found on websites that display Python package information and its history of versions, and often the package will have its own webpage. Given that we are crawling through every Python package, we needed a reliable way to locate every package, so we used pages that host all Python packages, such as pypi and libraries.io. From each package's page, factors being collected included stars, forks, watchers, contributers (Github), repository size, release date, and SourceRank, a score calculated by libraries.io's algorithm based on a number of metrics. Pulling from two sources prevented a considerable amount of NaN values, as it was often the case that one site had information on a factor that another did not. For the package's version, each version number and its timestamp for release time was scraped and placed into a list for the package.

Data Augmentation - How we have augmented it. We've got stats on GitHub repos, we've got info on lifetime of packages of PyPi & libraries.io

The agglomeration of raw data prompted a closer look into new factors taken from weighted combinations of existing data. Once again, the dependencies of packages would play a large role in influencing much of the operations during this

work. First, although we managed to find 5190 unique packages from the Binder launches, much of these packages were unfit to be used for analysis. Some were not installed properly when running the automated bash scripts, and some produced results that were not logically feasible, having extremely large sizes or installation times. The lack of information for some packages when crawling webpages, particularly on the package's version history, also prompted the removal of these from the overall dataset. After factoring in dependencies into a package's size and installation time requirements, such that each package's size and time correlates only to that single package (and not its dependencies as well), unneeded packages were filtered out, leaving a list of 4575 packages to work with. A list of full dependencies was connected to each package, such that a quick lookup could be made to find a package's dependencies and lookup for any input package. The updated package list was also applied to the specifications dataset.

A directed NetworkX graph with the libraries as the source node, the dependencies as the target nodes, and an arbitrary value as the weight was instantiated, as shown in Figure XX. In order to generate a meaningful graph, a threshold was supplied against the cosine similarity between any two nodes in the graph. As the threshold increased, the packages would have to be more similar in order for the edge to be kept in the graph. Using this data structure, additional factors were created, most notably the number and sum of the size of dependencies that filtered out of the overall data. Additionally, centrality measures were explored, and the authority value was chosen for analysis, representing the degree of incoming edges for a node.

Figure XX shows exploratory visualizations characterizing the relationships between factors in the dataset. In the heatmap, there are 12 columns/rows corresponding to different time metrics. Columns tagged with 'full' correspond to the installation time of a package and all its dependencies, while 'solo' denotes the time for just the installation of the package itself. 'Constant', which was not used later on, is the time to install the package when it was already installed (hence nothing being installed).

Our model also considers the compatibility between versions from two separate launches. The timestamp for each launch and the individual versions of each package were converted into a UNIX time format for numerical comparisons. In any recipe's specification, the packages required were optionally listed with a specific version. In the majority case where nothing was listed, the implication was that they wanted the most recent version, and this value was tagged onto the package for that launch by finding the highest UNIX time less than the time of the launch call.

Data Integration

Having discussed the methods for obtaining and augmenting the data, we now have the right data for building a model. The first is derived from the launches dataset, with the timestamp for the invocation of the Binder launch, the index needed to connect it to its specification, and a new column with a dictionary structure, each key being a package in the recipe and value being the version number required for that package. The second, which connects to the launches dataset, is the dependency table that was generated from the specifications dataset, where the rows and columns are the repositories (indices of the launches) and unique packages, and the value being the version number requested (which can easily be converted into binary values, where 1 represents existence). Finally, the sizes, installation times, dependency lists, version histories, and (Github) statistics are factors in a final table where the rows are all the unique packages.

Other notes

We've done manual experiments to get package install time and size.

Summary stats - Histograms - Repos run - Workload - Unique dependency sets (distribution) - Network graph

III. Simulator

What we've done with versions.

IV. Evaluation

No caching - show - Num of containers needed - Total time to create them (perhaps expressed as overhead on every invocation) - Total storage space for containers (over time)

LRU - Limits on sizes (number of containers or storage size?) - Strict versioning vs non-strict - Eval; hits, total size, total time

Reusing containers - Equality vs subsetting - Landlord (with alpha) - Eval; hits, total size, total time** (this is new)

Landlord ++ - Prioritizing evictions (maybe just on LRU) - E.g., lifetime score for a container based on all the packages in it, or a popularity score for all the packages.. - GitHub popularity - Container/package popularity based on training data - Lifespan of packages - Stretch: integrate the ones that seem useful into landlord - Stretch stretch: dynamic changing of strategy

V. Naive caching

VI. Naive caching

VII. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.