Network Analysis of Dependency Structure in Non-open Source Python Packages

Shan Zhong, Gaurav Gulati and Parth Pendurakar
University of California, Los Angeles

ABSTRACT

In this study, the structural dependency analysis technique was adopted to investigate the importance of non-open source python packages. More specifically, we applied models and algorithms to describe and analyze the dependencies among python packages from a network analytical perspective. The package network encodes interactions between two or more packages. The design and motivation for this study was driven by the findings in paper *Vertex Similarity in Networks* (Leicht, 2005). We simulated its method but replaced measure of similarity with measure of dependency. Multicore analysis is used to quantify the level of influence of each package community and to identify the distinctive ones.

1. INTRODUCTION

Software engineers and data scientists are constantly building and analyzing products to help companies grow. As a result, packages are built to encapsulate reusable code and build features at a faster rate. As new technology is built packages become dependent on other packages, which in turn creates a large network of applications being built off one another. For instance, most web applications have a front-end framework package to design the user interface of their product. This is a keynote as it creates competition for a marketplace of developing the best package for a certain product. For instance, companies like Google and Facebook have created their own front-end frameworks as a way to dominate in this space.

Software packages are continuously being created by the open source community as well big technology companies. Hence as more software is being created based on previously built packages the urgency for those packages to be reliant increases. In order to obtain a more in depth understanding of the network built by packages and their dependencies, we gathered data on all the python packages and a list of their respective dependencies. This felt like a good source for our analysis as python is one of the most prevalent languages used in current technology.

2. METHOD

2.1 Source Data Preprocessing

Our data comes from the Python Package Index (PyPI). PyPI gives a simple interface to query metadata about python packages and it contains 118,932 packages. Due to the limit of our computer's computing power, we only parsed 20000 dependencies data. Here, the dependency is defined as the relying relationship between other python packages and the given python package.

2.2 Exploratory Data Analysis

Figure 1 indicates the top python package types, top downloads of python package type and top large python packages. From the first table, we can conclude that the source distribution (sdist) whose results is an archive that can be used to recompile everything on any platform, is the largest package type, much larger than the second package type. However, the download statistics in the second table show that the package type setuptools has the largest usage rate reflected by its total number of downloads. Among all python packages in PyPI, the largest package is de422, as presented in the third table. The package contains the most recent long-period ephemeris published by the Jet Propulsion Laboratory.

The importance of packages is presented in figure 2. We adopted two ways to reflect package importance: one way is by calculating the number of connections for each package and the other way is to compute packages' PageRank scores. From the two resulting bar graphs, we can conclude that 'request', 'django', 'six', 'numpy', 'flask' and 'pyyaml' are the five most important packages in PyPI. The table inside figure 2 shows the ranking of non-system packages based on the number of module imports from the given packages.

2.3 Packages Dependency Analysis

To gain some insight into how the package hierarchies work from a network theoretical point of view, we need to analyze both the dependencies between packages, as well as dependencies within packages. Dependencies between packages are straightforward — if a package *x* depends on a package *y*, there will be an edge from the node representing the former to the node representing the latter. This form of dependency analysis is essentially a macroscopic perspective on the ecosystem of packages illustrated by the subset of overall Python packages we chose to take a closer look at. The result of such analysis can be seen in Figures 1-2, and 4-7. Dependencies within packages are a little more nuanced;

our analysis code inspects any given package by walking down relevant import module hierarchies and finding methods, functions, classes, built-ins, and other miscellaneous items that the said package depends on. Such analysis results in dependency structures within a package such as those shown in Figure 3 for *networkx*, *matplotlib*, and *pandas*, in particular.

2.3.1 Dependency between Packages

To perform dependency analysis between packages, we used a popular centrality measure, PageRank. PageRank, which was originally designed by Google as an algorithm to calculate the importance of web pages, is an extension of Katz centrality which measures the ranking of nodes in a graph based on the structure of incoming edges. PageRank outputs a probability distribution over the nodes (in this case packages), which can be interpreted as the likelihood that a package would need to be imported in any given piece of code. Furthermore, it is an iterative algorithm that converges to the theoretical ranking over infinitely many iterations.

While the premise is not so simple, it can be explained through with a walkthrough of how the algorithm would work. Assume we have *n* packages in the database that we are analyzing; the PageRank rankings (or probabilities) would initially be the same. That is, each of the *n* packages would have the same probability of being "chosen." The total importance, or *PageRank* of any given package is adjusted by a dampening factor and divided up by the number of outgoing edges and distributed to its dependents, over one iteration. This process continues for a set number iterations until a given tolerance threshold is reached.

We used PageRank to rank the importance of the packages in both the large dataset and our smaller dataset to get the results shown in Figure 2, as well as Figure 7. In Figure 7, we show the results of community analysis along with PageRank values for those given communities, which will be discussed later on.

From the graph in Figure 2, we see that *requests* holds the highest PageRank centrality, while packages such as *numpy* and *flask* follow closely behind. These results are not at all surprising, as we shall later see, *requests* also holds the top position for other centrality measures. Furthermore, libraries such as *numpy* and *flask* are widely used in many contexts, explaining the high PageRank associated with them in such a dataset. Furthermore, we also have to consider that other similar packages that were randomly dropped in our process of selecting a subset of data could have been right alongside with similar PageRank values, such as *pandas* with *numpy*.

2.3.2 Dependency within Packages (P)

To perform dependency analysis within packages, we wrote some code to walk through any given package and generate a dictionary of dependencies to draw a graph from. As aforementioned, the code inspects a package by iterating through imported module hierarchies recursively, tracking methods, functions, classes, built-ins, and other items that the said package has some sort of dependency or connection to. The pseudocode of our recursive algorithm follows is straightforward — for a package *x*, read in the package and iterate through the package's directory (a useful feature of Python). During the iteration, separate lists of modules, functions, classes, built-ins, and miscellaneous items are generated. Then, the function is called recursively on the list of modules, in the case where the submodule in question can be correctly imported and examined in a similar manner.

Upon running this function on three packages, *networkx*, *matplotlib*, and *pandas*, we generated Figure 3, the dependency graphs within each of the packages. In these graphs, blue represents a module dependency, purple a functions dependency, gold a class dependency, red a built-ins dependency, and black a miscellaneous dependency. As such, we see that each of the packages in question has one or two main modules that it depends on, with each of those in turn having several functions, classes, built-ins, and miscellaneous items that it is dependent on. We can also see the general layout of these packages at a high-level, giving us a sense of how many functions from how many modules it calls, for example. Looking at *networkx*, we see that there are two packages that it uses a multitude of functions from, given by the two large purple "circles" (referring to the way our code outputs the graphs) in the graph. On the other hand, *matplotlib* uses several items from a range of libraries, seemingly with a pretty even distribution in terms of functions and classes, with smaller purple and gold "circles" scattered around the main blue "circle". Finally, *pandas* calls several functions from two main subpackages, as we see many purple "circles" around the two large blue "circles". While this sort of analysis is more qualitative than quantitative, it still offers us a fairly good approximation of what the real distribution of dependencies in a package might look like.

2.4 Centrality Measures

There is no one specific answer to which package is the most important in the python community. Since there is no one ranking system for packages the answer to that question is a mix of opinions and the context of the conversation. Nonetheless, one can look at different centrality measures to get a better sense of what packages are most commonly used by developers. More importantly, with centrality measures one can take a look to see what packages have the most dependencies and create a paradox to see how engineers and data scientists work when a certain package breaks.

2.4.1 Degree Centrality

Given all the python packages a developer can access online, finding the packages with the highest degree centrality gives a great deal of information. For starters, the most intuitive item to look at are the packages that have the highest number of connections. For instance, the package with the highest number of connections is the *request* package. This makes sense as the *request* package helps developers send and receive HTTP requests from an API or other servers. It is noted that in the data that 3000+ packages are built off being dependent on the *request* package. As a result, the importance of keeping this package up to date without any bugs is heavily important. Given the data (118,932 packages) from the research collected roughly 4% to 5% of the packages in this network would be directly affected if the circumstances occurred where the *requests* package was to go down or have a default in an updated version. This does not include the ripple effect of the other packages that would get affected from the first-degree nodes and hence it is evident that degree centrality does give an important metric on which packages need to be working all the time. It is also important to note that the Page Rank Centrality also gives very similar results to the Degree Centrality as noted before. In both computations from the collected data, it is noted that the top three packages in order are *requests*, *django*, and *six*.

Another key aspect when discussing the degree centrality is what is the average degree for all packages in the python community. Based on the results, it seems that the data has a right skew with most packages having a very small degree. Nonetheless, there exists some hubs as talked about with packages like *requests* and *django*. The right skewed data shows a good sign when dealing with packages that have faulty updates. If the data was shown to have a left skew such that a majority of packages had a high dependency number this would be bad in the developer community because of the ripple effects. More specifically, an update with a bug will not only cause one package to be faulty, but can cause all the packages that depend on it to become prone to more bugs as well. It is also important to realize that one can not limit the number of packages with a high degree centrality as those specific packages are used to create some of the latest edge products for developers. For instance, companies like Spotify and Dropbox are using *django* as part of a key component in their tech stack. Hence it is good to keep these highly-dependent packages for innovation reasons, but the number of these packages should remain small to prevent entire networks of applications from crashing at the same time.

2.4.2 Betweenness Centrality

An important metric for analyzing which packages are the most important is the betweenness centrality. The betweenness centrality can be used as a method to quantify the number of times that a package in the network acts as a bridge along the shortest path between two other packages. This can be useful when analyzing different clusters in the network and what packages seem to be prevalent in multiple clusters. For instance, most developers write tests in parallel with writing the code for the actual application. This is due to the fact that unit and integration tests help determine how the application will work and makes it much faster to track down bugs as new iterations of the code are produced. As a result, based on the collection of data, the majority of the top ten packages with the highest betweenness connectivity are packages related to writing tests for different python applications. Also, similar to the degree centrality the package with the highest betweenness centrality is the *requests* package. This enhances the idea of how important the *requests* package is in the python community.

2.4.3 Closeness Centrality

Closeness centrality is another measure to check what packages are deemed the most important in the python community. The closeness centrality helps with creating a quantitative measure on what packages are relatively closer to all the other packages in the network relative to the other packages in the network. The measurement is influenced by what edges a node has and the total number of nodes in the network. As noted that the packages *requests* and *django* were popular in the developer community, the closeness centrality also backs up that claim. The research of the data collected shows that the packages *requests* and *django* are in the top ten for highest closeness centrality. However, instead of requests being the packages with the highest closeness centrality, it is noted that the *distribute* package is the package with the highest closeness centrality (followed up by packages *sphinx* and *six*). It is noted that the *distribute* package is used as a way for developers to easily build and distribute python packages, especially ones that have dependencies on other packages. As a result, a majority of any two packages in the network are connected by a shortest path that contains the *distribute* package because it was probably used to help publish the package to the open source community.

2.5 Package Communities

To generate the coordinate matrix graph, we used the Louvain community detection algorithm. This algorithm is a heuristic greedy approach to determining the best community partition based on increasing modularity incrementally. While it doesn't necessarily result in an optimal solution, it approximates it. It's important to note that the optimal solution is NP Hard, and thus, not necessarily worth exploring in an algorithmic context.

One important definition to note before understanding the algorithm is modularity. Modularity is a measure of how well divided a network is — networks that have high modularity are densely connected with respect to nodes in the same partition, and sparsely connection with respect to nodes in different partitions. The Louvain algorithm goes as follows: initialize every node to be its own partition. Then, until we don't have an increase in modularity, iterate through the nodes and join it with its neighbor in another partition if that join results in a higher modularity. This simple, yet effective heuristic can help us approximate the optimal solution for effective community detection.

Using the Louvain algorithm, we generated partitions of the subset of our data, and created a coordinate matrix graph by analyzing adjacency information for each partition sequentially. In the resulting graphs of the adjacency matrix, we see approximate blocks along the diagonal, which is expected. Each block represents a partition, with dense connections within, and only a few connections outside, resulting in several block-like shapes along the diagonal. Recall that this is analogous to a disconnected graph in which the nodes are ordered in such a way as to give the adjacency matrix the form of being the direct sum of the corresponding adjacency submatrices for each component.

3. FINDINGS

Figure 6 is the resulting graph of sparse adjacency matrix based on Louvain algorithm for community detection. The intersection between nodes (packages) is marked with a colored dot if there exists a connection between the packages. The set of largest squares located on the diagonal of the matrix are the identified communities. The size of the square is positively correlated to the number of packages inside each community and the density of the square is positively correlated to the number of complete connections inside each community. For instance, the square that contains less noise represents a community whose packages depend on nearly every other package.

In total, there are ten distinctive communities detected by the Louvain algorithm. Each community is marked with a number from 1 to 10 at the bottom of the coordinate matrix graph. The first community is centered on the flask and bottle packages. These are both web server frameworks, and provide an easy way to set up a web server. The second community has a lot of roughly equally-important nodes, which include redis, tornado, and pyzmq. Most of the packages inside the second community are required by ipython/jupyter notebooks. The third community is the pydata community. It is dominated by numpy, with smaller contributions from scipy, matplotlib, and pandas. The fourth community is dominated by testing and documentation packages and the fifth community is dominated by the django package, a framework for generating dynamic websites. The sixth community is a community for web interface and web scraping. The seventh community is the only community that can be visually discerned in the network graph and it is a zope community dominated by the distribute package. The eighth community is for web configuration and development since it contains packages such as jinja2 and pyyaml. The ninth community is a parser community which has utilities packages such as argparse, decorator, pyparsing and etc. The tenth community primarily serves for SQL queries since its most significant package is sqlalchemy.

Figure 7 provides the visualizations of the package dependencies inside the largest three communities and the most important packages inside those communities.

4. DISCUSSION

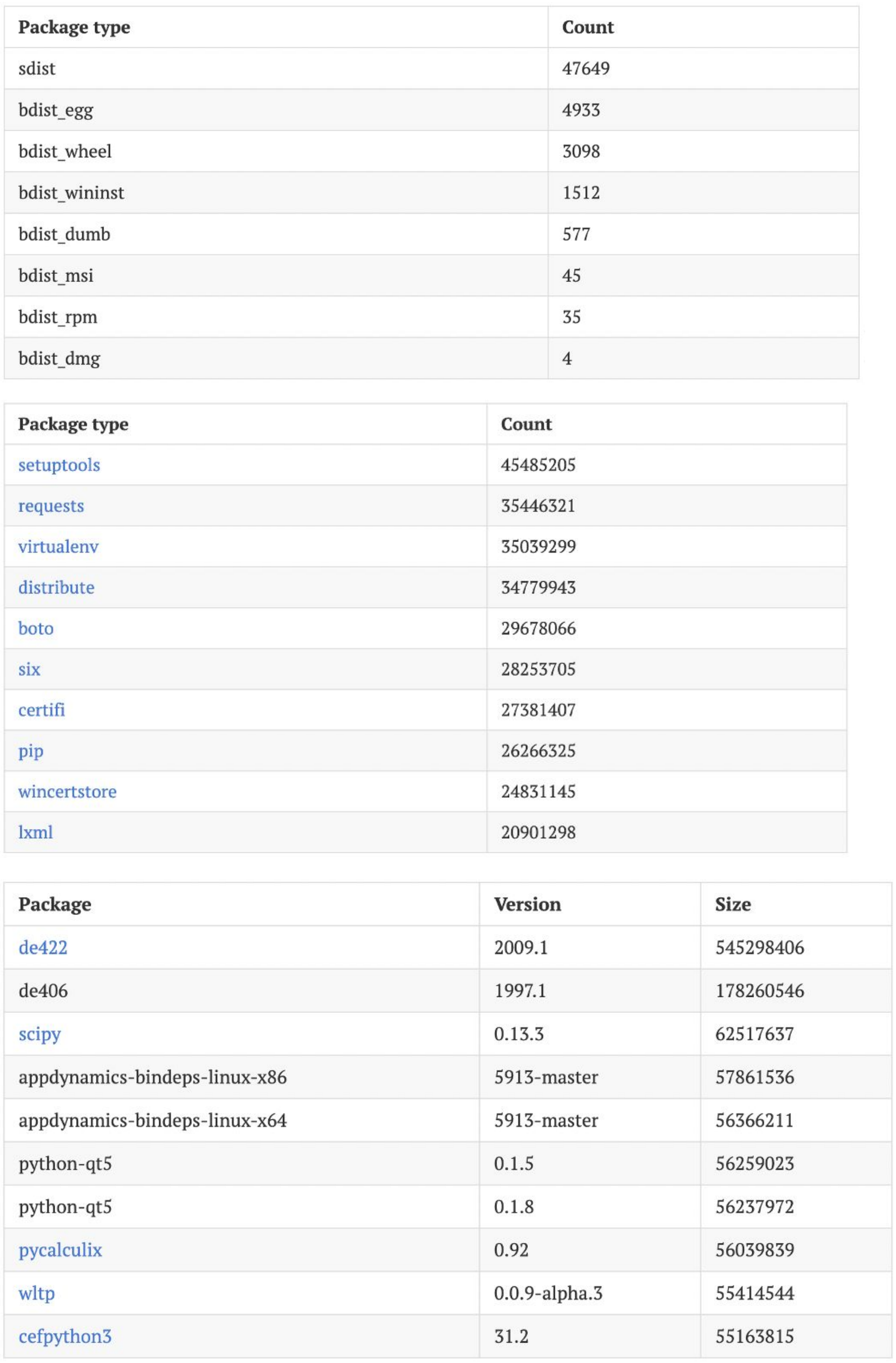
In the future, we hope to explore several other issues that came up during our analysis. Firstly, we would conduct these same experiments on the entire dataset, rather than just a small subset. This process would nevertheless require more brute computing power, as the algorithms used in the research scale poorly with larger and larger graph sizes. Secondly, we would like to explore the differences between choosing other damping factors for PageRank, as well as other parameters for the rest of the centrality measures. Perhaps this would give us more insight about how these dependencies truly behave, and allow for a more in-depth search for a custom centrality measure that would more accurately classify the "most important" packages — we can't assume that Python code packages ranks in an analogous manner to the way webpages do, for example.

Ultimately, with the evolving ecosystem of packages, versions, and other dependencies within Python and similar programming languages, it is difficult to truly quantify the importance and influence of non-open source packages. As a result, such experiments as the ones we have described in our paper should be repeated periodically, allowing researchers, mathematicians and programmers alike to garner more information about the packages that are relevant at the time.

5. REFERENCE

Data Source: https://pypi.org/

APPENDIX



Package type	Count
sdist	47649
bdist_egg	4933
bdist_wheel	3098
bdist_wininst	1512
bdist_dumb	577
bdist_msi	45
bdist_rpm	35
bdist_dmg	4

Package type	Count
setuptools	45485205
requests	35446321
virtualenv	35039299
distribute	34779943
boto	29678066
six	28253705
certifi	27381407
pip	26266325
wincertstore	24831145
lxml	20901298

Package	Version	Size
de422	2009.1	545298406
de406	1997.1	178260546
scipy	0.13.3	62517637
appdynamics-bindeps-linux-x86	5913-master	57861536
appdynamics-bindeps-linux-x64	5913-master	56366211
python-qt5	0.1.5	56259023
python-qt5	0.1.8	56237972
pycalculix	0.92	56039839
wltp	0.0.9-alpha.3	55414544
cefpython3	31.2	55163815

Figure 1: From top to down, each table describes top python package types, top downloads of python packages types and top large python packages. System packages were excluded from our calculation.

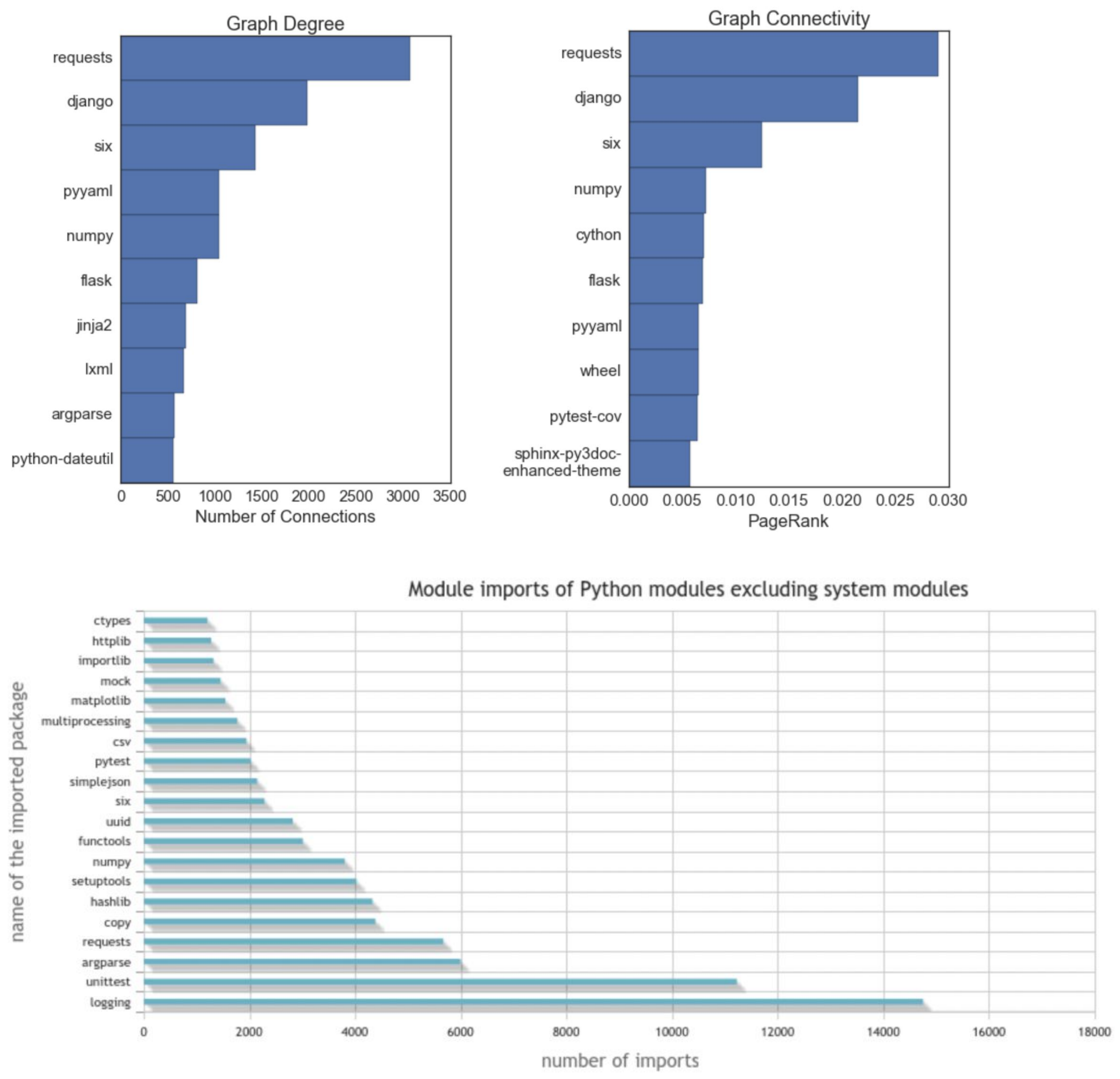


Figure 2: The bar graphs are for package importance and the table indicates the ranking of non-system packages based on the number of module imports from the given packages.

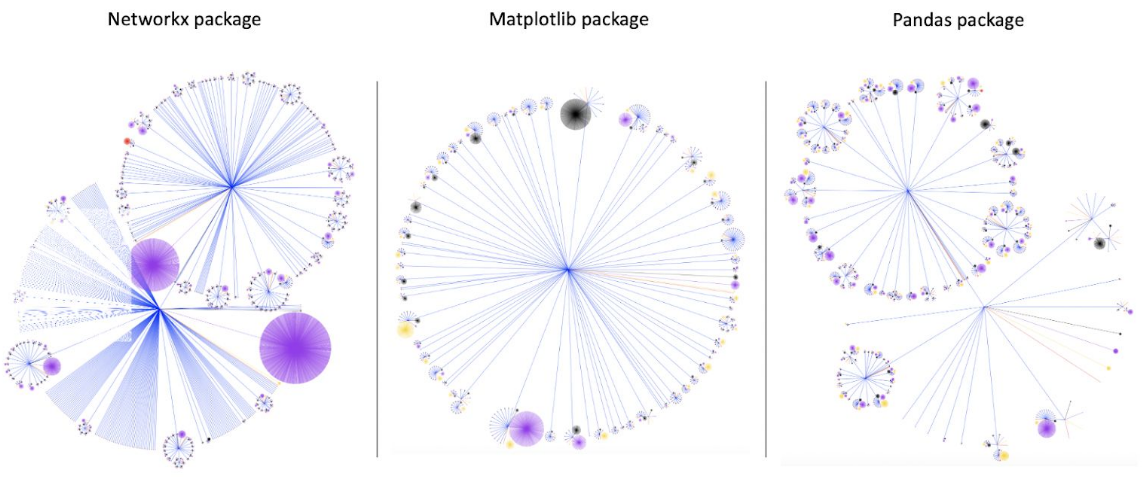


Figure 3: The dependency graph within three selected packagesc

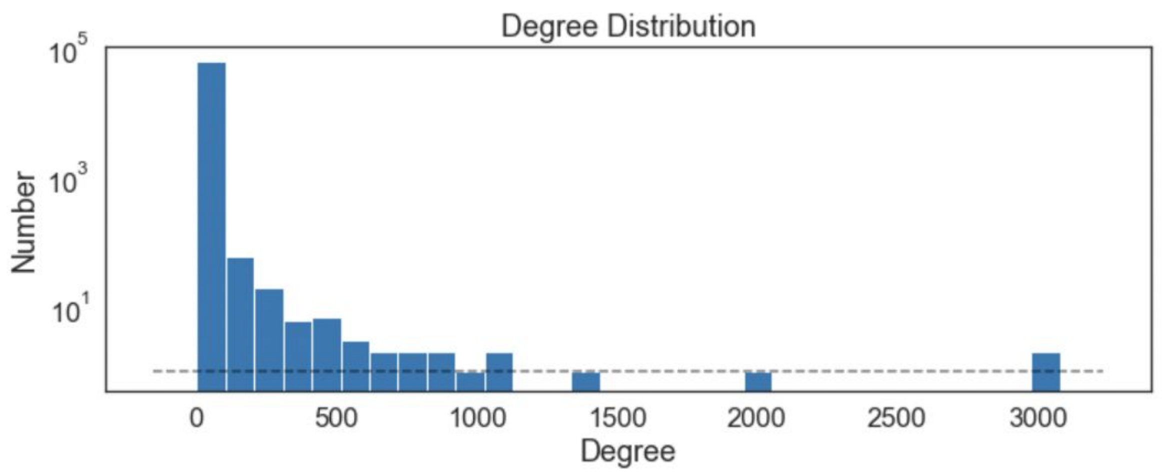


Figure 4: The graph of degree distribution for 20000 dependency data.

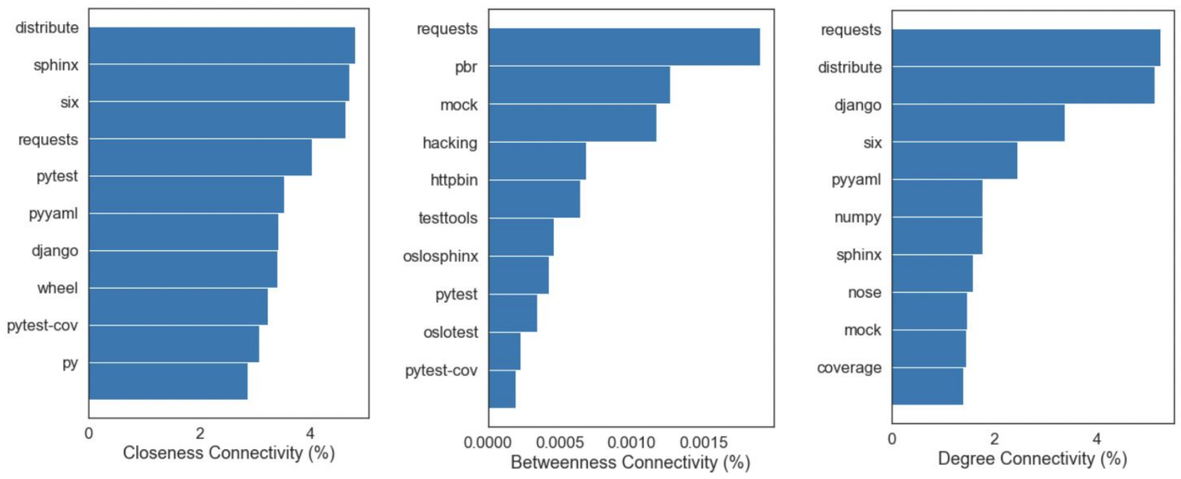


Figure 5 : Bar graphs for three centrality measures.

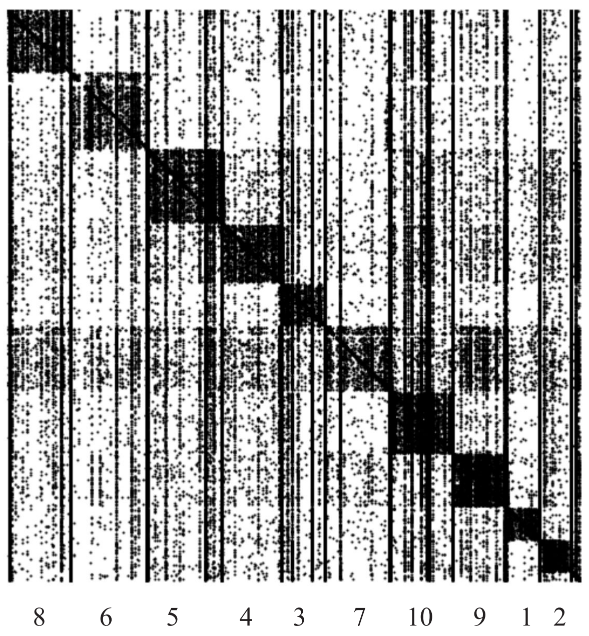
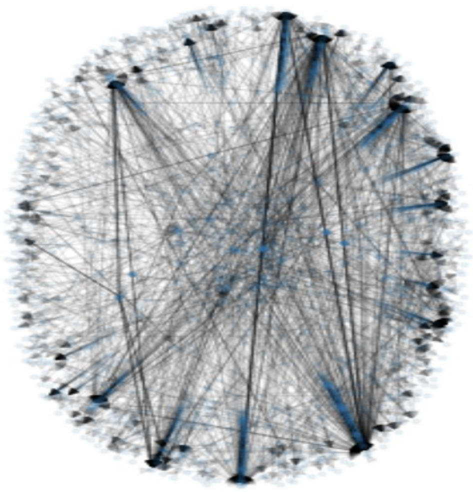
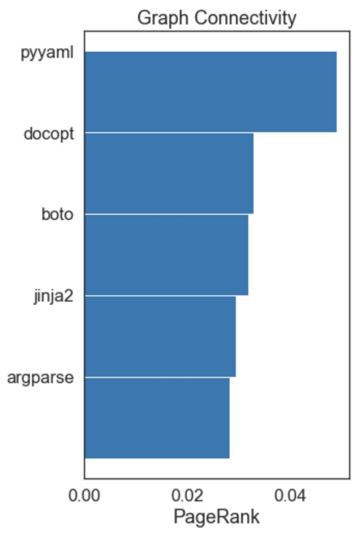
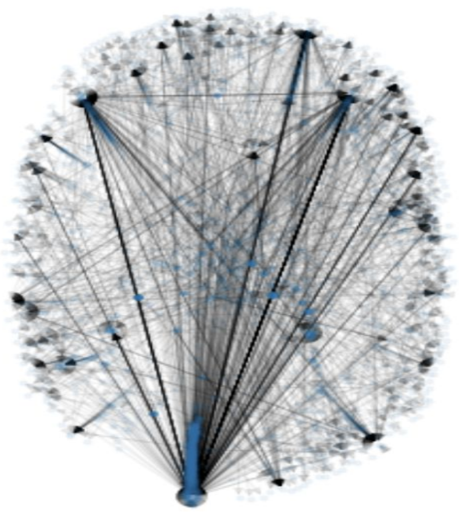
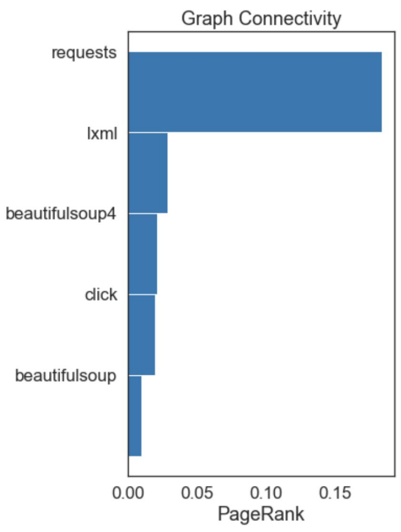


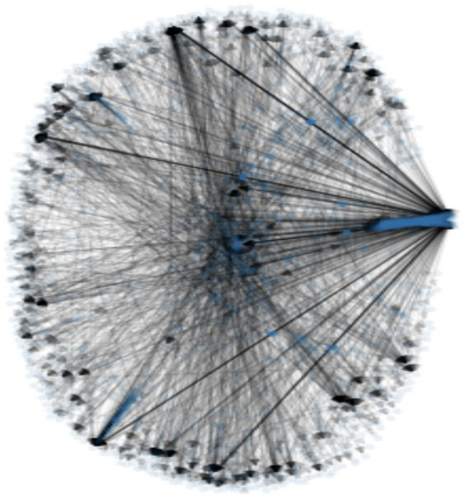
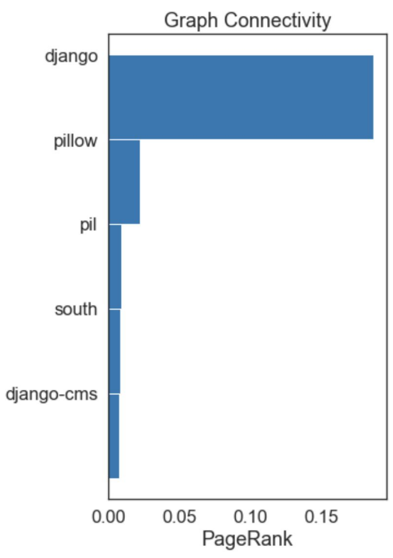
Figure 6: The resulting graph of sparse adjacency matrix based on Louvain algorithm for community detection.

alpha=0.08787

alpha=0.063

alpha=0.069

Figure 7: Visualizations of the package dependencies and most important packages inside the largest three communities.