Technology Adoption in Input-Output Networks

Xintong Han and Lei Xu*

Latest Version:
`http://leixu.org/Xu_JMP.pdf`

This Version: January 22, 2019

Abstract

This paper investigates the role of network structure in technology adoption. In particular, we study how the network of individual agents can slow down the speed of adoption. We study this issue in the context of the Python programming language by modeling the decisions to adopt Python version 3 by software packages. Python 3 provides advanced features but is *not* backward compatible with Python 2, which implies adoption costs. Moreover, packages form an input-output network through dependency on other packages in order to avoid writing duplicate code, and they face additional adoption costs from dependencies without Python 3 support. We build a dynamic model of technology adoption that incorporates the input-output network. With a complete dataset of package characteristics for historical releases and user downloads, we draw the input-output network and develop a new estimation method based on the dependency relationship. Estimation results show the average cost of one incompatible dependency is one-third the fixed cost of updating a package's code. Simulations show the input-output network contributes to 1.5 years of adoption inertia. We conduct counterfactual policies of promotion in subcommunities and find significant heterogeneous effects on the adoption rates due to differences in network structure.

*Han: Concordia University and CIREQ, Montreal, Canada. Email: xintong.han@concordia.ca. Xu: Toulouse School of Economics, Toulouse, France. Email: lei.xu@tse-fr.eu. We thank Daron Acemoglu, Victor Aguirregabiria, Luis Cabral, Allan Collard-Wexler, Jacques Cremer, Olivier De Groote, Pierre Dubois, Isis Durrmeyer, Daniel Ershov, Ana Gazmuri, Matthew Gentry, Paul Gomme, Gautam Gowrisankaran, Philip Haile, Doh-Shin Jeon, Thierry Magnac, Ariel Pakes, John Rust, Mario Samano, Paul Seabright, Elie Tamer, Robert Ulbricht, Kevin Williams, as well as seminar and conference participants at TSE, Concordia, Edinburgh, and EARIE for helpful comments. Financial support from TSE Digital Center and the NET Institute is gratefully acknowledged. All errors are our own.

1 Introduction

If one examines the history of the diffusion of many inventions, one cannot help being struck by two characteristics of the diffusion process: its apparent overall slowness on the one hand and the wide variations in the rates of acceptance of different inventions on the other.

– Rosenberg (1972)

Technological innovation has been one of the most important sources of productivity and economic growth (DeLong (2002), Mansfield, Mettler, and Packard (1980)), yet slow technology adoption is still a common phenomenon in many sectors (Geroski (2000)). The rich literature of technology adoption shows many factors that can slow down the speed of adoption: for example, organization (Atkin et al. (2017)), competition (Gowrisankaran and Stavins (2004)), and network effects (Saloner and Shepard (1995)).[1] This paper contributes to the literature by examining input-output networks, another important channel which can affect technology adoption.

As the economy becomes more disaggregated, firms specialize in one specific type of product or service and form input-output networks.[2] Specialization generates efficiency gains but linkages between firms may have adverse effects on new technology adoption (Katz and Shapiro (1985)). For example, wireless carriers are unable to adopt 5G technology without 5G equipment from suppliers. If suppliers experience negative shocks or are pessimistic about the new technology, 5G adoption overall may stall.[3]

The extent to which an input-output network can affect technology adoption depends on the network structure. Consider two examples of input-output networks as illustrated in Figure 1: Case (b) depicts a more disaggregated economy in which the intermediary firm 3 supplies its product to three other downstream firms. Thus, firm 3 in case (b) is more influential on technology adoption of the whole industry than in case (a). Policies that aim to promote a faster adoption rate need to consider the network of individual agents.

[1]For more examples, refer to the excellent surveys by Atkin et al. (2017), Hall and Khan (2003), and Hall (2009)

[2]An input-output network is also known in other literature as a vertical or hierarchical network.

[3]Other examples of technology adoption in input-output networks include payment systems and industry 4.0.

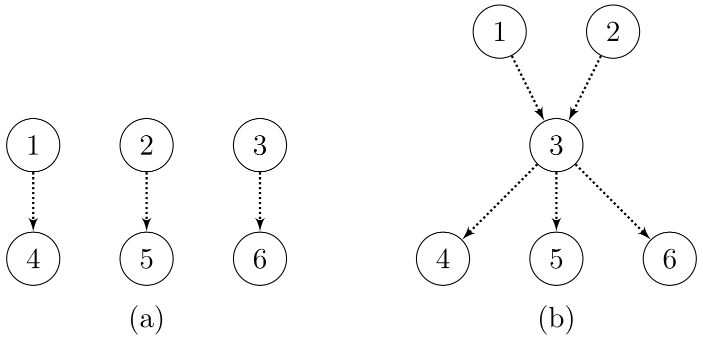


Figure 1: Examples of Input-Output Networks

This paper investigates how network structure affects individual technology adoption decisions. We study this issue in the context of the Python programming language, in particular, the transition from Python version 2 to version 3.[4] Python is one of the most popular programming languages in the world. Python 3 was a major release in 2008 that experienced slow adoption.[5] Python 3 provides several fundamental improvements but is *incompatible* with Python 2.[6] In other words, code written for Python 2 often fails to work on Python 3, and vice versa.

Like other programming languages, most functionalities on Python are provided by third-party packages. Packages are also known as libraries, (sub)routines, or modules in other programming languages. These packages form an input-output network through dependency requirements: Downstream packages are built using functionalities provided by upstream packages, or upstream packages are dependencies for downstream packages.[7] To use a package, the end user must install the package itself, as well as all of its upstream dependencies.

The incompatibility between Python 2 and 3 implies adoption costs for package developers. To use the new features of Python 3, packages need to update their source code. If one or more of their upstream dependencies have yet to adopt Python 3, additional adoption costs are incurred.[8] Our research focuses on the adoption decisions of third-party packages.

[4]Similar issues exist in other software or programming languages, such as Windows DLLs, Fortran, and Ruby. We study Python mainly due to data availability.

[5]The Python 3 adoption process has been widely considered a slower-than-optimal process.

[6]Appendix A provides a list of new functionalities in Python 3 that are incompatible with Python 2.

[7]Upstream packages are dependencies of downstream packages. The two terms "upstream" and "dependency" are used interchangeably throughout this paper.

[8]Typical solutions (or costs) include looking for alternative dependencies with Python 3 support, or updating the necessary components of the dependency in order to support Python 3.

We consider each package an individual agent making the adoption decision, and model utility as a function of user downloads (Fershtman and Gandal (2011)). For any commonly-known motivations behind contributions to Open Source Software (OSS) (e.g., altruism, career incentives, and ego gratification.), more user downloads are always better for package developers.

To understand how network structure affects the dynamics of technology adoption, we build a dynamic model in which each package makes an irreversible decision to adopt Python 3, following Rust (1987) and Keane and Wolpin (1997). This adoption decision implies changing the package to be compatible with Python 3. Our model highlights an intertemporal trade-off: If packages adopt early, they receive more user downloads over time but incur higher adoption costs; if packages adopt later, they may have lower adoption costs due to the future adoption decisions of upstream dependencies. The solution of the dynamic adoption model requires a prediction of future states for a package itself and for each of its dependencies, as well as those of the dependencies of dependencies, and so on.

With a complete dataset of package characteristics, historical releases and user download statistics,[9] we draw the input-output structure of all packages. We group packages into various layers based on the dependency relationship, and calculate the adoption probability layer by layer. Then we use maximum likelihood estimation (MLE) method to estimate the parameters.

Our model allows for rich heterogeneity across agents. Ryan and Tucker (2011) also measures the heterogeneous network effects, but mostly at the group level. Our paper incorporates a much richer level of heterogeneity at the individual agent level.

The variation in certain dependency characteristics affects utility in the future but not in the current period. We use such variation to identify the discount factor, which is typically not separately identified in many dynamic discrete choice settings.

Results from the structural estimation show packages benefit from more user downloads. We also show that upstream dependencies without Python 3 support pose significant barriers in the adoption decisions of downstream packages: The cost of dealing with one Python 3-

[9]Our data come from the Python Package Index project, which is the largest repository for Python packages. It records historical download information for more than 150,000 packages from 2005 onward.

incompatible dependency is equivalent to, on average, one-third of the fixed cost of updating their source code. We also show by simulation that such an input-output network of packages contributes to 1.5 years "excess inertia" in 2018.[10]

The structural model allows us to conduct counterfactual exercises of "sponsorship." A sponsor promotes the new technology, and can impact its future success (Katz and Shapiro (1986)). We evaluate the effectiveness of policies such as community-level targeted promotion of Python 3. Using modularity optimization tools developed in the social network literature, the packages are grouped into various "communities" based on the dependency network: for example, web development and data analysis communities. These communities differ significantly in their network structure. We test how technology adoption decisions can propagate in these communities. The results show that targeted community-level promotions have large heterogeneous effects on adoption rates due to differences in network structure within a community. Moreover, the promotion in one community has significant *positive or negative* effects on connected communities. The findings imply policies that consider network structure can significantly improve the effectiveness of promotion.

This paper contributes to the empirical literature by estimating the effect of network structure on technology adoption decisions.[11] Previous papers largely measured network effects using a "reduced-form approach" by modeling utility as the total number of users on the same network, and ignored the detailed linkages between individuals on a network. This paper also contributes to the literature of network analysis. Previous studies of networks mainly adopted static models, whereas in reality, individuals are inherently dynamic and face intertemporal tradeoffs. Failure to control for forward-looking agents can yield different estimation results and may misguide policymakers (Rust (1987), Hendel and Nevo (2006), Gowrisankaran and Rysman (2012)). This paper is among the first to link the literature of dynamic discrete choice models to network analysis. It is the first to show how input-output

[10]Excess inertia refers to slow adoption despite user benefits of new technology (Farrell and Saloner (1985)).

[11]Björkegren (2018) also studies technology adoption in a social network setting. He circumvents the complex problem by solving per-period equilibria of optimal timing to adopt, and assumes that each individual makes the adoption decision based on actual future adoption time of her contacts. Our approach uses the dynamic discrete choice model, which is more realistic and flexible: Individuals decide whether to adopt a new technology in the current time period, instead of when to adopt a new technology in a future time period; Agents are uncertain of others' adoption decisions, instead of having a perfect foresight of others' adoption decisions; The belief of others' adoption probability vary over time.

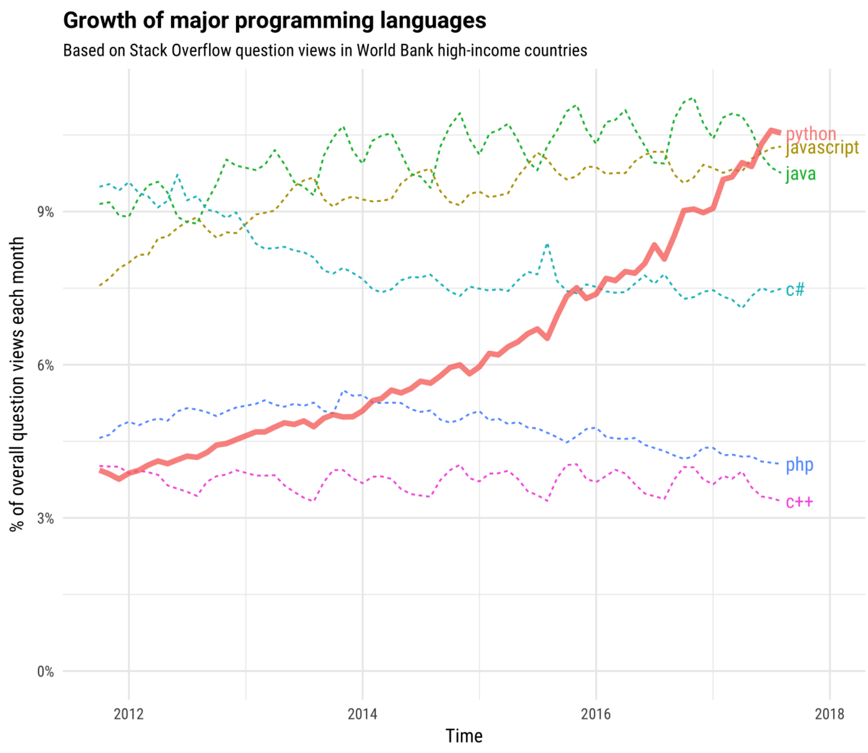


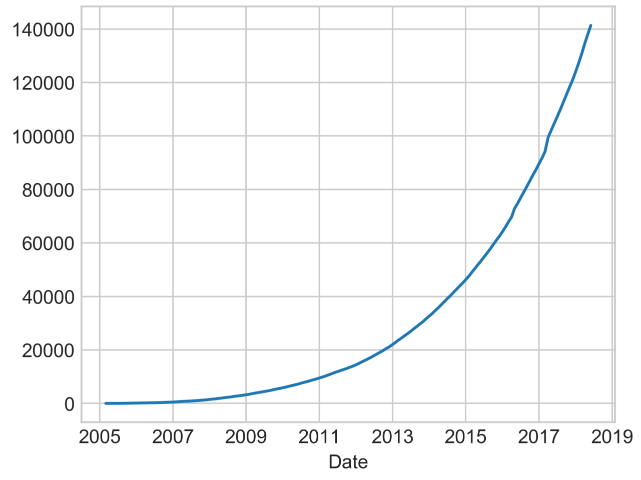
Figure 2: Popularity Trend of Programming Languages

networks can lead to adverse effects on the technology adoption process in a dynamic setting.

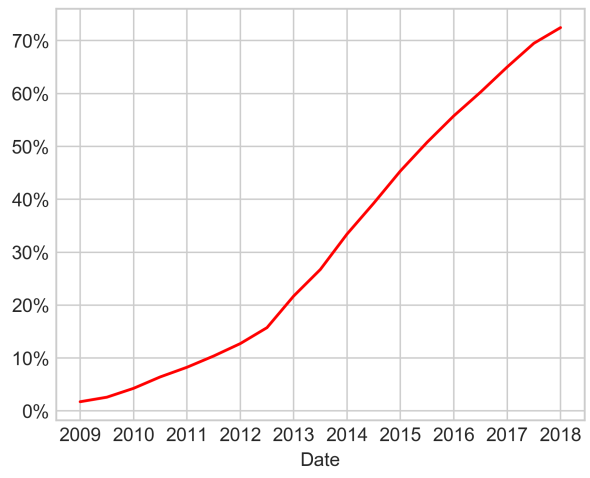
2 Background: the Python Programming Language

Python is a general-purpose programming language.[12] It has a syntax that allows users to express concepts in fewer lines of code compared to most other programming languages, and has been widely used in introductory computer science courses at universities. The first version was released in 1989 but did not gain popularity until the late 2000s. In the past few years, Python has become the fastest-growing major programming language. Figure 2 plots the numbers of visits to questions related to a particular programming language on Stack Overflow, the largest Q&A website for programming-related matters. Python has grown to be the number one based on this measure.

[12]A general-purpose programming language is a computer language that is broadly applicable across application domains. Examples of general-purpose programming languages include C, Java, and Python. It is in contrast to domain-specific language, such as MATLAB (numerical computing), Stata and R (statistical analysis), etc.



(a) Total Number of Packages



(b) % of Packages with Py3 Support

Figure 3: Python Packages with Python 3 Support

Backward compatibility has been a widely disputed topic in the software industry. The tradeoff is clear: easier user transition to new technology vs. higher cost of development and slowing innovation. In order to introduce several key features to Python, the core developers decided to break the backward compatibility with the major release of Python 3 in 2008.[13] Users and package developers who want to run their code on Python 3 have to confirm that all the source code is compatible with Python 3.[14]

The transition process has been longer than many expected. Due to a large existing Python 2 user base, many packages are reluctant to provide Python 3 support. For those who do, they usually provide two versions for each release: one for Python 2 and another for Python 3.

Figure 3 plots the proportion of packages that provide Python 3 support. It has experienced steady growth, during which time the total number of packages has also been growing exponentially, indicating the growing popularity of the Python programming language in general.

[13]Python core developers are a group of most active contributors to the Python programming language, which itself is an OSS. Some of the major new features of Python 3 include newer classes, Unicode encoding, and float division. Please refer to `http://python.org/` for more detailed information. Several examples of incompatibility are shown in Appendix 11.1.

[14]Some packages are developed to help users to transit to Python 3 easier automatically. However, in most cases, users and package developers still have to test and manually modify much of code.

Table 1: Package Characteristics

name	statsmodels
license	BSD
summary	Estimation and inference for statistical models
author	Josef Perktold, Chad Fulton, Kerby Shedden
version	0.9
requires_dist	numpy
	pandas
	matplotlib
classifiers	Intended Audience :: Science/Research
	Programming Language :: Python :: 2
	Programming Language :: Python :: 3
	Topic :: Scientific/Engineering

2.1 Motivation of Package Developers

Both the Python programming language and almost all third-party packages are Open Source Software (OSS).[15] The motivations of OSS contribution by software developers can be multifold.[16] The literature on motivations for private contributions to public goods suggests that the most common motivations are altruism, career concerns, ego gratification, etc. For any type of motivations, developers always prefer more user downloads. Therefore, we assume that the payoff of package developers as a function of user downloads.[17]

3 Data

We collect data from the Python Package Index (PyPI), a repository of software for the Python programming language. In other words, it is a website where developers upload their packages so that users can find and download packages they need.

When uploading packages to PyPI, package developers usually provide various information related to the files, such as the description of the package, the contact information of owners, whether they provide support for Python 2 or Python 3, and what other packages

[15]Nearly all the Python packages in our study are open source, which are free of charge to anyone to use. A limited number of packages offer free downloads but requires payment for usage.

[16]von Krogh et al. (2012) and Xu, Nian, and Cabral (2016) provide overviews of the literature.

[17]A similar assumption has been used in other papers in the study of OSS (see Fershtman and Gandal (2011) and Fershtman (2016).

Table 2: Downloads Statistics

(a) Before 2016: Cumulative Download

upload_time	2014-12-02
python_version	3.4
downloads	41564
filename	statsmodels-0.6.whl
size (bytes)	3969880

(b) After 2016: Individual Download

timestamp	2018-09-01
country_code	FR
filename	statsmodels-0.6.whl
project	statsmodels
version	0.6
python	3.4
system	Mac OS X

are required as dependencies. Table 1 lists some basic information related to a package. The dependency requirement comes from the "requires_dist" section, and the Python 2/3 support status comes from the "classifiers" section.[18]

User downloads data consists of two separate sources, both recorded by PyPI. Before 2016, cumulative download statistics for each file is recorded (see Table 2.a). The system stopped working for a few months in early 2016 until May 2016, when PyPI introduced a new system hosted on Google BigQuery.[19] The new system records and publishes certain information related to each individual download (as shown in Table 2.b). We combine the two sources of user downloads data and extrapolate the number of downloads for the missing time periods from January to May 2018.

Although there are currently more than 150,000 packages hosted on PyPI (as of September 2018), the vast majority are either abandoned or hobby projects for personal use. For example, 40% of all packages have only one or two releases. Figure 4 plots the total downloads of top N packages as a percentage of all downloads in 2017. It shows a long tail of downloads across packages: top 100 packages account for approximately 58% of all downloads and top 500 packages account for 79% of all downloads.

For our analysis, we focus on packages that are well-maintained with regular releases. Thus, we select packages based on the following criteria (values in parenthesis are the unconditional percentage of packages that satisfy each measure):

[18]In addition to the developer-provided information in the package description section, we also extract dependency requirements from the source files of each package and infer Python 2/3 support from filenames.

[19]Google BigQuery provides cloud-based data warehouse services. It can record a large amount of data in real time, and provides end users with easy access and manipulation of the stored data.

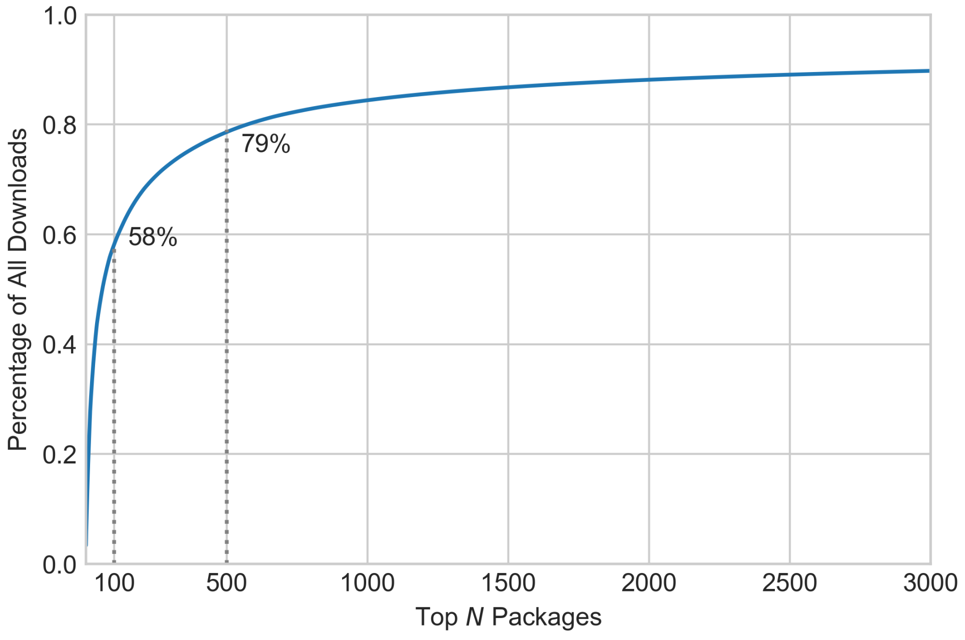


Figure 4: Total Downloads of Top Packages as Percentage of All Downloads

- Time Duration (Last Release - First Release Date) $\geq$ 1 Year (12.9%)
- Downloads per Year $\geq$ 2000 (30.8%)
- Total Number of Releases $\geq$ 5 (38.9%)
- Total Releases / Time Duration $\geq$ 1 (92.4%)
- Some Python 2/3 Support Info Available (59.9%)

These selection criteria provide 4,003 packages and 13,056 observations for our analysis. Other selection measures are used for robustness checks.

Table 3 shows the summary statistics of several key variables between the whole population and our selected sample. The selected sample includes the majority of the most popular packages on Python, consisting 51.69% of all of the downloads on PyPI. The average number of downloads for each package in the selected sample is also much higher than the whole population. On average, the selected sample has 3.12 dependencies vs. 0.72 in the whole population. These packages are also larger in file size. Consistent with other figures in this

Table 3: Summary Statistics

	All	Selected
Number of packages	143,584	4,005
Total downloads (in percentage)	100.00%	51.69%
Average logarithm of downloads per package	5.87	11.51
(min, max)	(0.0, 19.3)	(8.8, 19.3)
Average number of dependencies	0.72	3.12
(min, max)	(0, 174)	(0, 79)
Average logged package size (MB)	2.78	4.19
(min, max)	(0.0, 13.2)	(0.5, 10.6)

section, Table 3 implies that our analysis focuses on the well-maintained packages with regular releases that faced a adoption decision of Python 3. These packages are also the most popular packages in the Python community.

All the decision and state variables in our model are measured in six-month periods. For example, 2013/01/01 to 2013/06/30 is one period, and 2013/07/01 to 2013/12/31 is another.

4 Input-Output Networks

With a complete dataset of Python packages and characteristics, we draw the whole input-output network of packages through their dependency requirements. Such an input-output network can also be represented by an adjacency matrix $\mathcal{G} \in \{0,1\}^n$, a $n \times n$ matrix where n equals to the number of packages in the network. For each element $g_{i,j} \in \mathcal{G}$, $g_{i,j} = 1$ if package j is the dependency of package i, and $g_{i,j} = 0$ otherwise. We assume in our paper that $\mathcal{G}$ is time-invariant and is commonly known by packages.

Figure 5 plots a sample of dependency network. The arrows represent the dependency relationship. For example, An arrow from *NumPy* to Scipy means *Numpy* is the dependency of *SciPy*.[20] In this case, we also call *NumPy* the upstream package to the downstream package *SciPy*. The adjancency matrix of the network shown in Figure 5 can be written as

[20]In principle, the whole input-output network is acyclic. In other words, circular dependency relationship such as $A \to B \to C \to A$ is not supposed to exist. In the data, there exist a small number of cases of circular dependencies. We compare the characteristics of all package pairs and manually remove the weakest link, measured by the number of times a package is used as a dependency.

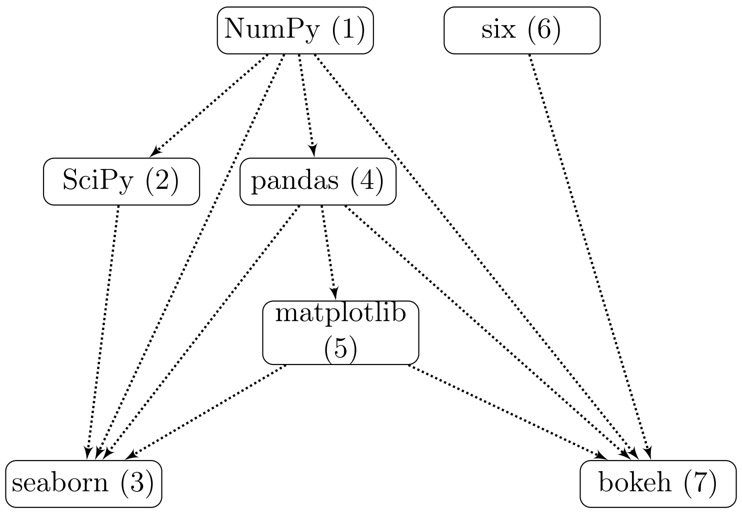


Figure 5: Example of the Input-Output Network of Packages

the following:

$$\mathcal{G}_{Fig5} = \begin{array}{c|ccccccc|} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}.$$

4.1 Layered Network Representation

The elements in the adjacency matrix $\mathcal{G}$ can be re-arranged to various sets in order to represent the acyclical dependency relationship. We denote the set of dependencies of package i as $\mathcal{U}_i = \{j|g_{i,j} = 1\}$. Then we group packages $i = 1, 2, 3, \cdots, n$ to L sets: $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \cdots, \mathcal{L}_L$, where for any package $i \in \mathcal{L}_l$, $\mathcal{U}_i \subseteq \mathcal{L}_{1,2,\cdots,l-1}$ where $\mathcal{L}_{1,2,\cdots,l-1} = \mathcal{L}_1 + \mathcal{L}_2 + \cdots + \mathcal{L}_{l-1}$. The packages can be categorized into layers using the following algorithm:

$$\mathcal{L}_1 = \{i \mid |U_i| = 0\}$$

$$
\begin{aligned}
\mathcal{L}_2 =& \{i \mid U_i \subseteq \mathcal{L}_1\} \\
\mathcal{L}_3 =& \{i \mid U_i \subseteq \mathcal{L}_{1,2}\} \\
&\vdots \\
\mathcal{L}_l =& \{i \mid U_i \subseteq \mathcal{L}_{1,2,\cdots,l-1}\} \\
&\vdots \\
\mathcal{L}_L =& \{i \mid U_i \subseteq \mathcal{L}_{1,2,\cdots,L-1}\}.
\end{aligned}
$$

where $\mathcal{L}_{1,2,\cdots,l} = \mathcal{L}_1 + \mathcal{L}_2 + \cdots + \mathcal{L}_l$

Then the adjacency matrix $\mathcal{G}$ can be written using block matrices:

$$
\mathcal{G} = \begin{bmatrix}
0 & 0 & \cdots & 0 & 0 \\
H_{2,1} & 0 & \cdots & \cdots & 0 \\
H_{3,1} & H_{3,2} & \ddots & \ddots & \vdots \\
\vdots & \vdots & \ddots & 0 & 0 \\
H_{L,1} & H_{L,2} & \cdots & H_{L,L-1} & 0
\end{bmatrix},
$$

where each block matrix $H_{l,l'}$ represent the adjacency matrix between packages in layer l and l'. The diagonal and upper triangular block matrices are zeros by definition that all dependencies of any package are included in upper layers.

Figure 6 depicts layered network representation using the previous example in Figure 5.

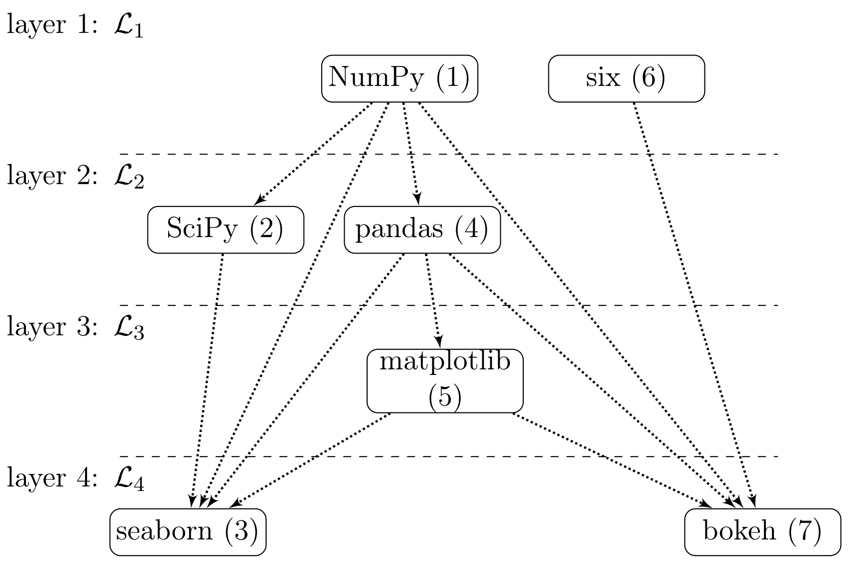


Figure 6: Example of the Layered Network Representation

The associated adjacency block matrix can be written as:

$$\mathcal{G} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ H_{2,1} & 0 & 0 & 0 \\ H_{3,1} & H_{3,2} & 0 & 0 \\ H_{4,1} & H_{4,2} & H_{4,3} & 0 \end{bmatrix} = \begin{array}{c|cc|cc|c|cc} & 1 & 6 & 2 & 4 & 5 & 3 & 7 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 5 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 3 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 7 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \end{array},$$

where $L = 4$, $\mathcal{L}_1 = \{1, 6\}$, $\mathcal{L}_2 = \{2, 4\}$, $\mathcal{L}_3 = \{5\}$, $\mathcal{L}_4 = \{3, 7\}$.

4.2 Communities in Python

Python is a general-purpose programming language, meaning that it is not designed for a specific community of users; rather, it is designed in a flexible way so that users from any domains can customize the language for their own use (e.g., data analysis and web development). The needs of communities are met by third party packages, most of which are designed to achieve certain tasks in a specific field. Those packages that serve users within a given community tend to be more closely linked through dependencies.

We use an heuristic method of modularity optimization (?) to extract 10 major communities within the Python community, based on the input-output network of Python packages. Denote $\mathcal{M}_m$ as the set of packages in community m. Then the adjacency matrix $\mathcal{G}$ of the network can be re-written as:

$$\mathcal{G} = \begin{bmatrix} H_{1,1} & H_{1,2} & \cdots & H_{1,M} \\ H_{2,1} & H_{2,2} & \cdots & H_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ H_{M,1} & H_{M,2} & \cdots & H_{M,M} \end{bmatrix},$$

where each block matrix $H_{m,m'}$ represent the adjacency matrix between packages in community m and m'. The dependency relationship among packages within a given community m is captured by $H_{m,m}$. These packages are more closely linked to each other than pacakges across different communities, captured by $H_{m,m'}$ where $m \neq m'$.

Figure 7 plots the transformed adjacency matrix that uses community-based block matrices. It shows the dependency network relationships both within and across the top 10 communities. Therefore, each black square represents a community; a larger square means more packages in that community; a darker (or dense) one means packages in that community are more closely linked to each other.

In Section 9, we provide more detailed discussion on these communities, and test the effectiveness of Python 3 promotion across different communities.

5 Model

Our model is based on the single agent dynamic choice framework developed by Rust (1987) to analyze technology adoption decisions by packages of the Python programming language. Each package i at time t can be described by a state variable $\mathcal{S}_{i,t}$. Given the current state $\mathcal{S}_{i,t}$, each package i makes an irreversible decision to adopt Python 3, namely,

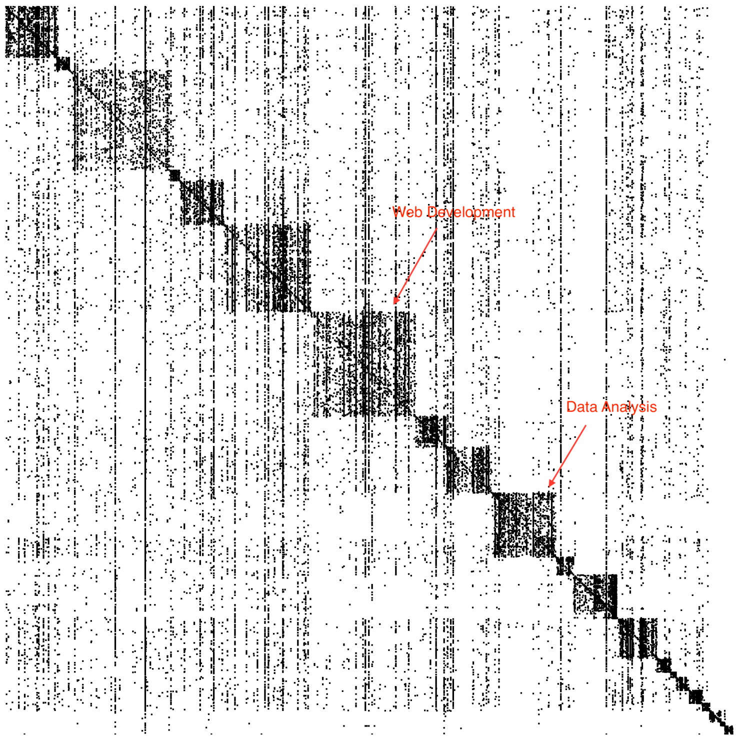


Figure 7: Python Community Through Dependency Network

to make the package compatible with Python 3.[21] Let $d_{i,t}$ be the a binary irreversible decision:

$$d_{i,t} = \begin{cases} 1 & \text{if package } i \text{ adopts Python 3} \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Packages are linked through dependency requirements. In order to incorporate the dependency network of packages and how packages affect each other through their adoption decisions, the state variable includes both one's own characteristics as well as those of dependencies. Thus, the state variable can be specified as the following:

$$\mathcal{S}_{i,t} = \Bigg\{ \underbrace{x_{i,t-1}}_{\substack{\text{user}\\\text{downloads}}}, \underbrace{z_{i,t}}_{\substack{\text{other package}\\\text{characteristics}}}, \underbrace{\epsilon_{i,t}^{d_{i,t}}}_{\substack{\text{i.i.d.}\\\text{shocks}}}, \underbrace{d_{i,t-1}}_{\substack{\text{previous}\\\text{adoption decision}}}, \underbrace{\{S_{j,t}, d_{j,t}\}_{j \in U_{i,t}}}_{\substack{\text{states and decisions}\\\text{of dependencies}}} \Bigg\}.$$

[21]More examples of irreversible decisions as well as discussion can be found in Rust and Phelan (1997) and Aguirregabiria and Mira (2010).

The dependency network among packages is incorporated in the last component of the state variable. One implicit assumption is a sequential game play, that is, package i observes the adoption decision made by its dependencies. Section 5.3 provides more discussion and implications on this assumption.

All adoption decisions are irreversible, meaning that in each time period t, only packages without Python 3 support make the adoption decisions. The adoption decision comes with a one-time adoption cost, and this affects the transition dynamics of the state variable $\mathcal{S}_{i,t}$.

5.1 Value Function and Bellman Equation

Given the state $\mathcal{S}_{i,t}$, a package's flow utility can be written as:

$$\begin{aligned} & u(S_{i,t}, d_{i,t}; \theta) - C(S_{i,t}, d_{i,t}; \theta) + \nu_{i,t}^{d_{i,t}} && (2) \\ =& u(S_{i,t}, d_{i,t}; \theta) - \mathbf{1}(d_{i,t-1} = 0, d_{i,t} = 1)\, C(S_{i,t}; \theta) + \nu_{i,t}^{d_{i,t}}, && (3) \end{aligned}$$

where $u(S_{i,t}, d_{i,t}; \theta)$ is the reward function of the indirect utility, which is a function of user downloads as discussed in Section 5.2. $C(S_{i,t}; \theta)$ is the one-time cost to adopt Python 3.

The value function for packages without Python 3 support can be written as the following dynamic problem:

$$\begin{aligned} & V_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 0) && (4) \\ =& \max_{\{d_{i,t+\tau}\}_{\tau=0}^{\infty}} \mathbf{E}_t\{\sum_{\tau=0}^{\infty} \beta^\tau \Big(u(S_{i,t+\tau}, d_{i,t+\tau}; \theta) - D_{i,t+\tau}\, C(S_{i,t+\tau}; \theta) + \nu_{i,t+\tau}^{d_{i,t+\tau}}\Big) | \mathcal{S}_{i,t}; \theta\}. && (5) \end{aligned}$$

The Bellman equation implies at each period t, the ex ante value function, conditional on $d_{i,t-1} = 0$, can be represented by:

$$\begin{aligned} & V_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 0) && (6) \\ =& \max_{d_{i,t} \in \{0,1\}} u(S_{i,t}, d_{i,t}; \theta) - d_{i,t}\, C(S_{i,t}; \theta) + v_{i,t}^{d_{i,t}} + \beta \mathbf{E}_t V_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t}) && (7) \\ =& \max\{v_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 0, d_{i,t} = 0), v_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 0, d_{i,t} = 1)\} && (8) \\ =& \max\{u(S_{i,t}, d_{i,t} = 0; \theta) + v_{i,t}^0 + \beta \mathbf{E}_t V_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t} = 0), && (9) \end{aligned}$$

$$u(S_{i,t}, d_{i,t} = 1; \theta) - C(S_{i,t}; \theta) + v^1_{i,t} + \beta \mathbf{E}_t V_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t} = 1)\}. \tag{10}$$

The representation of value function for packages with Python 3 support is more straightforward, since the irreversibility condition implies that no further decisions are being made:

$$V_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 1) \tag{11}$$

$$= v_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 1, d_{i,t} = 1) \tag{12}$$

$$= \mathbf{E}_t\{\sum_{\tau=0}^{\infty} \beta^\tau \Big(u(S_{i,t+\tau}, d_{i,t+\tau} = 1; \theta) + \nu^1_{i,t+\tau} \Big) | \mathcal{S}_{i,t}; \theta\} \tag{13}$$

$$= \sum_{\tau=0}^{\infty} \beta^\tau \int \Big(u(S_{i,t+\tau}, d_{i,t+\tau} = 1; \theta) + \nu^1_{i,t+\tau} \Big) dF_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}). \tag{14}$$

We assume that $v^{d_{i,t}}_{i,t}$ are iid errors that follow Type 1 Extreme Value Distribution. Then the expected value function $\mathbf{E}_t V_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t})$ can be calculated using the following equation:

$$\mathbf{E}_t V_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t} = 0) \tag{15}$$

$$= \int V_\theta(\mathcal{S}_{i,t+1}, d_{i,t} = 0) dF_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t} = 0) \tag{16}$$

$$= \int \log\{ \sum_{d_{i,t+1} \in \{0,1\}} \exp(v_\theta(\mathcal{S}_{i,t+1}, d_{i,t} = 0, d_{i,t+1}))\} dF_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t} = 0) \tag{17}$$

$$\mathbf{E}_t V_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t} = 1) \tag{18}$$

$$= \int V_\theta(\mathcal{S}_{i,t+1}, d_{i,t} = 1) dF_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t} = 1) \tag{19}$$

$$= \sum_{\tau=1}^{\infty} \beta^{\tau-1} \int \Big(u(S_{i,t+\tau}, d_{i,t+\tau} = 1; \theta) + \nu^1_{i,t+\tau} \Big) dF_\theta(\mathcal{S}_{i,t+1} | \mathcal{S}_{i,t}, d_{i,t} = 1). \tag{20}$$

Given the parameter θ and the transition dynamics described by F_θ, EV is iterated until convergence, which is then used to calculate choice-specific value functions $v_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 0, d_{i,t})$. The choice-specific value functions allow us to compute the predicted probability of adopting Python 3 using the standard logit formula:

$$P(d_{i,t} = 1|\mathcal{S}_{i,t}, d_{i,t-1} = 0; \theta) = \frac{v_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 0, d_{i,t} = 1)}{\sum_{d' \in \{0,1\}} v_\theta(\mathcal{S}_{i,t}, d_{i,t-1} = 0, d')} \tag{21}$$

$$P(d_{i,t} = 0|\mathcal{S}_{i,t}, d_{i,t-1} = 0; \theta) = 1 - P(d_{i,t} = 0|\mathcal{S}_{i,t}, d_{i,t-1} = 0; \theta) \tag{22}$$

$$P(d_{i,t} = 1|\mathcal{S}_{i,t}, d_{i,t-1} = 1; \theta) = 1 \tag{23}$$

$$P(d_{i,t} = 0|\mathcal{S}_{i,t}, d_{i,t-1} = 1; \theta) = 0. \tag{24}$$

5.2 Payoff Function

As discussed in Section 2.1, we model the utility of package developers as a function of user downloads. Denote $x_{i,t}$ as the logrithm of total number of times a package i is downloaded by users in period t. We specify the payoff function as a linear function of user downloads:

$$u(S_{i,t}, d_{i,t}; \theta) = \alpha^x \, x_{i,t}(x_{i,t-1}, d_{i,t}; \theta). \tag{25}$$

The evolution of $x_{i,t}$ relies on the package's adoption status/decision $d_{i,t}$. We assume that the evolution of the demand following a first-order Markov process is:

$$x_{i,t} = \rho_0 + \rho_r \cdot d_{i,t} \cdot r_t + \rho_1 \cdot x_{i,t-1} + \epsilon_{i,t} \quad with \tag{26}$$

$$d_{i,t} = \begin{cases} 1 & \sum_{\tau \leq t} d_{i,\tau} = 1 \\ 0 & \sum_{\tau \leq t} d_{i,\tau} = 0 \end{cases}, \tag{27}$$

where d indicates if the package i has been adopted before $t + 1$, r_t is the current Python 3 adoption rate among all packages, $\upsilon^0_{i,t}$ and $\upsilon^1_{i,t}$ are two white noises that are normally and interdependently distributed with mean 0 and variances $\sigma^2_{d_0}$ and $\sigma^2_{d_1}$.[22]

The AR1 evolution of user demand is estimated outside the model of Python 3 adoption decisions by package developers. Then the estimates are used as inputs to the structural

[22]In the estimation of the user downloads function, we assume that package developers can perfectly predict the future values of r_t.

model.

The estimates of the AR1 demand equation essentially serve as the counterfactual demand for a package deciding whether to adopt Python 3 or not. However, the estimates of the AR1 process may suffer from an endogeneity problem. The AR1 process is estimated using the evolution of user downloads in a result of the actual adoption decision. The packages that have adopted Python 3 may experience a positive persistent demand shock that is unobservable to the econometrician. In other words, due to the endogeneity issue, the benefit of Python 3 adoption inferred by the AR1 estimates can be over-exaggerated.

In order to control for the endogeneity, we introduce a two-step estimation approach using a synthetic instrumental variable. Given an initial set of AR1 estimates, we first estimate the model of technology adoption. Then in the second step, we use the predicted adoption probability as an instrumental for the endogenous variable $d_{i,t}$ and re-estimate the AR1 process. Then the new estimates are fed back to step 1 to re-estimate the model of technology adoption. Step 1 and 2 are repeated until all of the estimates converge to a fixed point. Further discussion on identification can be found in Section 6.1.

5.3 Adoption Cost

We model the cost function as a function of agent's decision $d_{i,t}$, the number of coding lines (internal cost) $code_i$ and the network constraint $\mu_{i,t}$ that represents the number of dependencies without Python 3 support at time t, adjusted by an "importance" measure of that dependency. The switching cost is thereby defined as below:

$$C_{i,t} = AC_0 + \alpha^{\mu}\,\mu_{i,t}, \tag{28}$$

and we further assume that $\mu_{i,t}$ admits the following representation:

$$\mu_{i,t} = \sum_{j \in \mathcal{U}_i} \mathbf{1}\{d_{j,t} = 0\}. \tag{29}$$

$\mu_{i,t}$ is built as a raw measure of the adoption cost due to incompatible dependencies. In reality, such a barrier may differ across different dependencies. For example, it may be easier to find a dependency alternative for another small dependency compared to a large one. In the estimation stage, we also include $\mu_{i,t}^{adj} = \sum_{j \in \mathcal{U}_i} \mathbf{1}\{d_{j,t} = 0\} z_{j,t}$, that is, $\mu_{i,t}$ adjusted using the characteristics of the dependencies.

Assumption 1. *At time t, a package i observes Python 3 adoption decisions made by its dependencies, namely, $d_{j,t}$ for all $j \in U_{i,t}$.*

This is an assumption on the information set available to a package when making decisions. In particular, the question is whether package i observes the adoption decisions of its dependencies $j \in U_{i,t}$ before making its own adoption decision. If yes (as in Assumption 1), then it's more appropriate to model a sequential move game; if not, then a simultaneous-move game would be more appropriate.

Assumption 1 simplifies our model estimation but only slightly. In our model, the adoption cost comes from the dependencies without Python 3 support. At time t, a package i weighs the tradeoff between adopting today versus wait for a later time, taking into consideration of the adoption probability of each of the Python 3-incompatible dependencies. Compared to a simultaneous-move version, the set of dependencies without Python 3 support at time t under the current assumption is weakly smaller, which can alleviate the computational burden in cases where both the upstream and downstream packages adopt Python 3 in the same time period, and such cases comprises a small percentages of cases in the data.

We prefer the sequential-move assumption due to the following reasons: First, upstream packages tend to be more popular packages who often pre-announce their plans for future releases, including the Python 3 adoption decisions. Second, a downstream package is likely to pay close attention to decisions made by its dependencies because it directly depends on them in order to work. Thus, it is likely that the Python 3 adoption decisions of upstream packages are readily available to downstream packages as soon as they are made.

Assumption 1 implies a sequential-move game where upstream packages make decisions first, followed by the downstream packages. The exact order of play in the model is defined based on the network structures, and it will be specified in detail in later sections.

We do not think that a simultaneous-move game (without Assumption 1) and a sequential one (with Assumption 1) would produce significantly different results. The only observations that can give rise to different estimates are cases when a package and its dependencies adopt Python 3 in the same time period, which does not represent a large share of cases in the data.

If package i observes $d_{j,t} = 1$, then the perceived adoption cost is much smaller than the case without observing $d_{j,t}$. In this case, the cost parameter with a simultaneous-move model is smaller compared to a sequential-move version.

Assumption 2. *A package i does not explictly consider responses from its downstream packages k where $i \in U_{k,t}$.*

A package i cares about the responses from its downstream packages insofar as it cares about more user downloads of its own package. The mean effect is captured and approximated by the parsimonious AR1 process of user downloads.[23] Assumption 2 implies that a package does not explicitly include the reaction function of its downstream packages in its utility function. Without explicitly modeling the interaction of upstream and downstream packages, assumption 2 significantly reduces the computational burden.

We believe that Assumption 2 is reasonable because upstream packages tend to experience significantly more downloads than their downstream counterparts, i.e., indirect downloads through downstream packages often comprise a small portion of total user downloads. Moreover, an upstream package often has many downstream packages and it's unlikely to track all the downstream packages dependent on it. Neither the econometrician nor the packages have a clear knowledge regarding the portion of downloads coming from direct versus indirect channels.[24]

5.4 State Variables & Transition Probability

As mentioned in the previous sections, one important component of the adoption cost comes from the dependency network when the dependency packages lack Python 3 support,

[23]Future versions of equation 26 will also include package characteristics and/or package fixed effects.

[24]Nonetheless, in the future versions of this paper, we plan to integrate downstream packages' responses in $u_{i,t}$ using methods developed in the dynamic games literature, namely BBL. Similar to the AR1 process of user downloads, BBL also serves as an approximation of the responses of downstream packages without explicitly modeling the response function.

which is represented by the last component of the state variables $\{d_{j,t}, S_{j,t}\}_{j\in U_{i,t}}$. We model a dynamic model of sequential decisions where upstream packages make adoption decisions before downstream packages, and downstream packages observe the decisions made by upstream packages at time t, namely $d_{j,t}$.

Package i cares about $\{d_{j,t}, S_{j,t}\}_{j\in U_{i,t}}$ in so far as it cares about its own adoption cost (a function of $\mu_{i,t}$), as well as its future evolution based given the current states. The law of motion of $\mu_{i,t}$ can be computed in the following way:

$$\begin{aligned} &\mathbf{P}(\mu_{i,t+1} = \mu' | \mu_{i,t} = \mu, \{d_{j,t}, \mathcal{S}_{j,t}\}_{j\in\mathcal{U}_{i,t}}; \theta) \\ =\ &\mathbf{P}(\mu' - \mu = \Delta\mu | \{d_{j,t}, \mathcal{S}_{j,t}\}_{j\in\mathcal{U}_{i,t}}; \theta), \end{aligned} \tag{30}$$

where $\Delta\mu$ is the change in the number of packages lacking Python 3 support from time t to $t{+}1$ and $\Delta\mu \in \{0, -1, -2, ..., -\mu\}$, that is, the adoption cost in future periods might be lower. Again, $\{d_{j,t}, S_{j,t}\}_{j\in U_{i,t}})$ is important in predicting dependency j's probability of Python 3 adoption at a future date. Denote package i's belief that its dependency j will adopt Python 3 at time $t{+}1$ as $\widehat{p}^1_{j,t+1} = \mathbf{P}(d_{j,t+1} = 1 | d_{j,t} = 0, \mathcal{S}_{j,t}; \theta)$, and $\widehat{p}^0_{j,t+1} = \mathbf{P}(d_{j,t+1} = 0 | d_{j,t} = 0, \mathcal{S}_{j,t}; \theta)$.

Then we can write equation 30 as:

$$\mathbf{P}(\mu' - \mu = \Delta\mu | \{d_{j,t}, \widehat{p}_{j,t+1}\}_{j\in\mathcal{U}_{i,t}}; \theta). \tag{31}$$

With $\mu_{i,t} \equiv \sum_{j\in U_{i,t}} \mathbb{1}(d_{j,t} = 0)$, we can further simplify 31 by denoting the set of dependencies without Python 3 support as $\Omega_{i,t} \equiv \{j \in U_{i,t} | d_{j,t} = 0\}$, thus $\mu_{i,t} = |\Omega_{i,t}|$.

Then equation 31 can be written in the following way:

$$\mathbf{P}(\mu' - \mu = \Delta\mu | \{\widehat{p}_{j,t+1}\}_{j\in\Omega_{i,t}}; \theta). \tag{32}$$

The adoption decisions by different packages in $\Omega_{i,t}$ can lead to the same value of $\Delta\mu$. Define $\mathcal{O}_{i,t}$ as the power set of $\Omega_{i,t}$ that contains all possible subsets of $\Omega_{i,t}$. Further we denote $\mathcal{O}^k_{i,t} = \{o \in \mathcal{O}_{i,t} \bigm| |o| = k\}$, that is, all the elements of set $\mathcal{O}_{i,t}$ with the same cardinality of k.

Then equation 32 can be written as:

$$\sum_{o' \in \mathcal{O}_{i,t}^{\mu'}} \mathbf{P}(o' \mid \Omega_{i,t}, \{\widehat{p}_{j,t+1}\}_{j \in \Omega_{i,t}}; \theta) \tag{33}$$

$$= \sum_{o' \in \mathcal{O}_{i,t}^{\mu'}} \left(\prod_{j \in o'} \widehat{p}^{0}_{j,t+1} \prod_{j \in \Omega_{i,t} \setminus o'} \widehat{p}^{1}_{j,t+1} \right). \tag{34}$$

The burdensome mathmatical notation can be more easily understood with a simple example. Suppose that at time t, package i has three dependencies $U_{i,t} = \{A, B, C\}$, and $d_{A,t} = 0$, $d_{B,t} = 0$, $d_{C,t} = 1$, leaving the set of dependency without Python 3 support as $\Omega_{i,t} = \{A, B\}$. Suppose package i's belief that each of the dependencies will adopt Python 3 at time $t+1$ with the following probability: $\widehat{p}^{1}_{A,t+1} = a$, $\widehat{p}^{0}_{A,t+1} = 1 - a$, $\widehat{p}^{1}_{B,t+1} = b$, $\widehat{p}^{0}_{B,t+1} = 1 - b$. The powerset of $\Omega_{i,t}$ is $\mathcal{O}_{i,t} = \{\varnothing, \{A\}, \{B\}, \{A, B\}\}$, and $\mathcal{O}^{0}_{i,t} = \{\varnothing\}$, $\mathcal{O}^{1}_{i,t} = \{\{A\}, \{B\}\}$, $\mathcal{O}^{2}_{i,t} = \{\{A, B\}\}$. Following equation 34, the transition from $\mu = 2$ to $\mu' = 1$ can be calculated as:

$$\sum_{o' \in \mathcal{O}^{1}_{i,t} = \{\{A\},\{B\}\}} \left(\prod_{j \in o'} \widehat{p}^{0}_{j,t+1} \prod_{j \in \Omega_{i,t} \setminus o'} \widehat{p}^{1}_{j,t+1} \right) \tag{35}$$

$$= \prod_{j \in \{A\}} \widehat{p}^{0}_{j,t+1} \prod_{j \in \{A,B\} \setminus \{A\}} \widehat{p}^{1}_{j,t+1} + \prod_{j \in \{B\}} \widehat{p}^{0}_{j,t+1} \prod_{j \in \{A,B\} \setminus \{B\}} \widehat{p}^{1}_{j,t+1} \tag{36}$$

$$= \widehat{p}^{0}_{A,t+1} \cdot \widehat{p}^{1}_{B,t+1} + \widehat{p}^{0}_{B,t+1} \cdot \widehat{p}^{1}_{A,t+1} \tag{37}$$

$$= (1 - a)\, b + a\, (1 - b). \tag{38}$$

5.5 Transition Matrix

The calculation of EV in equation 17 depends crucially on the specification of the transition matrix, or $\mathbf{P}_{\mathcal{S}'|\mathcal{S}}$ in equation 17.

In our model, the utility function is mainly governed by two variables, namely, $x_{i,t}$, the measure of downloads or popularity, and $\mu_{i,t}$, the measure of adoption cost due to dependencies. The construction of the transition matrix depends on the joint law of motion of $x_{i,t}$ and $\mu_{i,t}$.

The law of motion of $x_{i,t}$ is relatively easy. We assume that it follows an AR1 process as specified in equation 26, with the parameter values estimated outside the value function iteration.

On the other hand, the law of motion of $\mu_{i,t}$ is much more difficult. $\mu_{i,t} \equiv \Omega_{i,t} \equiv \sum_{j \in U_{i,t}} \mathbb{1}(d_{j,t} = 0)$, that is, the number of dependencies of package i without Python 3 support at time t.

One of the most important tradeoffs for package i to adopt Python 3 today at t vs. future periods $t + \tau$ is the decreasing adoption cost due to the decreasing number of dependencies without Python 3 support over time. Therefore, the solution to the dynamic adoption model depends on the calculation of future adoption probabilities for each of the dependencies.

The calculation is a formidable task due to the nature of the nested dependency network. This computational difficulty can be illustrated by forecasting the Python 3 adoption probability for each of package i's dependency $j \in U_{i,t}$ at time $t + 1$:

$$\widehat{p}^1_{j,t+1} = \int_{\mathcal{S}_{j,t+1}} \mathbf{P}(d_{j,t+1} = 1 | \mathcal{S}_{j,t+1}, d_{j,t} = 0, z_j; \theta) d\mathbf{P}(\mathcal{S}_{j,t+1} | \mathcal{S}_{j,t}, d_{j,t} = 0, z_j; \theta). \tag{39}$$

The integral can then be computed through a simulation of future states $\mathcal{S}_{j,t+1}$. Let $\mathcal{S}^m_{j,t+1}$ be the mth simulation, then the value of $\widehat{p^1_{j,t+1}}$ can be obtained by:

$$\widehat{p}^1_{j,t+1} = \frac{1}{M} \sum_{m=1}^{M} \mathbf{P}(d_{j,t+1} = 1 | \mathcal{S}^m_{j,t+1}, d_{j,t} = 0, z_j; \theta). \tag{40}$$

This simulation method can be very computationally intensive. Note that the nested states are expressed as $(x_{i,t-1}, d_{i,t-1}, \nu_{i,t}, \{d_{j,t}, S_{j,t}\}_{j \in U_{i,t}})$. The simulation of the states of package j requires the simulation of the states of j's dependency $k \in U_{j,t+1}$, as well as the states of k's dependency, and so on. Any slight changes in any of the linked dependencies, or the dependencies of each dependency, can affect $p_{j,t}$. In this way, the full solution approach by Rust (1987) is no longer feasible due to the curse of dimensionality problem. In fact, with the 3,102 packages and 13,056 observations, it is simpler to build the transition matrix dynamically for each package i at each time period t. In later Section 6, we list the detailed steps outlining how to compute $p_{j,t}$ for $j \in \Omega_{i,t}$ given the input-output network.

A transition matrix used for the dynamic programming problem includes the transition probability, not only from the current state to the next, but also from each of the possible states to all other states. Given the state $S_{i,t}$, package i calculates $\widehat{p}^{1}_{j,t+\tau}$ for each of $j \in \Omega_{i,t}$ and $\tau \in \mathbb{N}$.

Given the transition probability specified in equation 34, the full transition matrix needed to calculate $EV(S, d = 0; \theta)$ can be specified as the following:

$$P(o' \in O_{i,t} | o \in O_{i,t}) = \begin{cases} 0 & \text{if } o \nsubseteq o' \\ \prod_{j \in o'} \widehat{p}^{0}_{j,t} \prod_{j \in o \setminus o'} \widehat{p}^{1}_{j,t} & \text{if } o \subseteq o' \end{cases}. \tag{41}$$

The calculation of the transition matrix, as specified in equation 41, can be illustrated using the same example in Section 5.4. Recall that the set of dependencies without Python 3 support is $\Omega_{i,t} = \{A, B\}$. The adoption probability of A and B at time t can be calculated by package i. Assume that $\widehat{p}^{1}_{A,t} = a$ and $\widehat{p}^{1}_{B,t} = b$. Assumption **??** implies that $\widehat{p}^{1}_{A,t+\tau} = a$ and $\widehat{p}^{1}_{B,t+\tau} = b$ for all $\tau \in \mathbb{N}$. The powerset of $\Omega_{i,t}$ is $\mathcal{O}_{i,t} = \{\varnothing, \{A\}, \{B\}, \{A, B\}\}$, and $\mathcal{O}^{0}_{i,t} = \{\varnothing\}$. Therefore, the transition matrix of μ can be calculated using equation 41:

$$TM(\mu_{i,t}) = \begin{array}{c} \\ \{A,B\} \\ \{A\} \\ \{B\} \\ \varnothing \end{array} \begin{array}{c} \begin{array}{cccc} \{A,B\} & \{A\} & \{B\} & \varnothing \end{array} \\ \begin{bmatrix} (1-a)(1-b) & a(1-b) & (1-a)b & ab \\ 0 & 1-a & 0 & a \\ 0 & 0 & 1-b & b \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}.$$

The corresponding value of μ for each row (or column) is 2,1,1,0, respectively. By construction, the transition matrix considers both the number and the identities of dependencies that are without Python 3 support. For example, the situation with dependency A being the only one without Python 3 support differs from the situation when B is the only one. Both cases have the same value of μ which is 1, thus having the same adoption cost, but the value of waiting is different because A and B have different likelihoods of adoption in future periods. The transition matrix calculated using equation 41 considers of all such cases.

6 Identification & Estimation

6.1 Identification

In this section, we provide a brief discussion about the identification of our structural parameters.

The law of motion of x is modeled as a first-order Markov process, and it can be identified from the variation of x_t over time. Following the existing literature of dynamic discrete choice models (Keane (1994), Keane and Wolpin (1997)), we estimate $AR(1)$ separately from the structural model of technology adoption for computational purposes.

Once the law of motion of $x$is determined, one can identify the fixed cost AC_0, α^{μ} through the predicted variation of $\Delta\mu$ as a function of (μ, x, d, θ).

In most settings of dynamic discrete choice models, the discount factor is typically assumed, and is not able to be separately identified from other parameters (Magnac and Thesmar (2002)). The identification of the discount factor requires variations that can shift expected discounted future utilities, but not current utilities (Abbring and Daljord (2018), De Groote and Verboven (2018)). We are able to identify the discount factor through the differences in dependency characteristics, which affect future values but not current utility. For example, two packages with the same characteristics and the same number but different dependencies have the same utility in the current period. However, the different dependencies have different future adoption probabilities, thus a different transition matrix. If the packages are myopic ($\beta = 0$), the two packages with different dependency characteristics should behave the same way; if the packages are forward looking ($\beta > 0$), larger differences in dependency characteristics should have a larger impact on the differences between the adoption probabilities of the two packages.

By assuming all packages share the same discount factor β, the variations of downloads allow us to identify α^{x}.

6.2 Estimation

Our model of technology adoption is estimated using the Maximum Likelihood Estimation (MLE). The likelihood function is defined as:

$$l(\theta) = \prod_{i=1}^{N} \prod_{t=1}^{T_i} \widehat{p}_{i,t}^{0} \,^{\mathbf{1}\{d_{i,t}=0\}} \widehat{p}_{i,t}^{1} \,^{\mathbf{1}\{d_{i,t}=1\}},$$

where $\widehat{p}_{i,t}^{1} \equiv \widehat{p}^{1}(S_{i,t};\theta)$, which is defined in equation 21.

6.3 Parameters

The parameters of interest in our model are the following:

$$\theta = \{\underbrace{\rho_0, \rho_1, \rho_r}_{\theta_D};\ \underbrace{\alpha^x, AC_0, \alpha^\mu, \alpha^{size}, \beta}_{\theta_S}\}.$$

It is convenient to group the parameters into θ_D and θ_S. θ_D includes parameters for the demand function, which is modeled as an AR1 process of user downloads; θ_S includes parameters for the supply side, which is the model of technology adoption.

6.4 Layered Input-Output Network

The adoption decision of a package i depends crucially on the adoption status of its dependencies, which is summarized as $\mu_{i,t}$ in the utility function. Further, Assumption 1 states that $\mu_{i,t}$ is known to package i before making decisions at t. It allows us to model the adoption decisions sequentially and calculate Python 3 adoption probability for packages layer by layer (see Section 4.1). In every time period t, packages in layer 1 decide to adopt Python 3, followed by those in layer 2, then those in layer 3,$\cdots$, etc. The probability of adopting Python 3 for each of the upstream packages is then used as given for the decision making process of the downstream package, in order to calculate the transition probability of $\mu_{i,t}$.

6.5 Estimation Procedure

Our algorithm begins by setting initial values for θ_D, which comes from OLS estimation of the AR1 process of user downloads, specified in equation 26. Estimation then involves iteration on the four steps, where the m^{th} iteration follows:

Step 1: Estimation of the Model of Technology Adoption

- Given estimates of demand function θ_D^m and an initial guess of $\theta_S = \{\alpha^x, \alpha^\mu, AC_0\}$:
  - for $i \in \mathcal{L}_1$, build transition matrix for each i, t and calculate $\widehat{p}^1(\mathcal{S}_{i,t}; \theta)$
  - for $i \in \mathcal{L}_2$, given $\widehat{p}^1(\mathcal{S}_{j,t}; \theta) \quad \forall j \in \mathcal{L}_1$, build transition matrix for each i, t and calculate $\widehat{p}^1(\mathcal{S}_{i,t}; \theta)$
  - for $i \in \mathcal{L}_3$, given $\widehat{p}^1(\mathcal{S}_{j,t}; \theta) \quad \forall j \in \mathcal{L}_{1,2}$, build transition matrix for each i, t and calculate $\widehat{p}^1(\mathcal{S}_{i,t}; \theta)$

    $\vdots$
  - for $i \in \mathcal{L}_l$, given $\widehat{p}^1(\mathcal{S}_{j,t}; \theta) \quad \forall j \in \mathcal{L}_{1,2,\cdots,l-1}$, build transition matrix for each i, t and calculate $\widehat{p}^1(\mathcal{S}_{i,t}; \theta)$

    $\vdots$
  - for $i \in \mathcal{L}_L$, given $\widehat{p}^1(\mathcal{S}_{j,t}; \theta) \quad \forall j \in \mathcal{L}_{1,2,\cdots,L-1}$, build transition matrix for each i, t and calculate $\widehat{p}^1(\mathcal{S}_{i,t}; \theta)$
- Calculate likelihood fuction $l(\theta)$
- Update θ such that $\theta_S^{m+1} = \underset{\theta_S}{\operatorname{argmax}}\, l(\theta_S, \theta_D^m)$.

Step 2: Re-estimate Demand Using IV

- Given θ_D^m and θ_S^{m+1}, calculate the predicted adoption probability $\widehat{p}^1(\mathcal{S}_{i,t}; \theta_D^m, \theta_S^{m+1})$ for all i, t
- Re-estimate the AR1 process of demand function using $\widehat{p}_{i,t}^1$ as IV for $D_{i,t}$, and get a new set of estimates θ_D^{m+1} using the standard IV estimation method.

7 Results

Table 4: Demand Estimation (AR1 Process)

	(1) OLS	(2) IV
(ρ_r) $d_{i,t} \times r_t$	0.165***	0.074***
	(0.01)	(0.01)
(ρ_1) $x_{i,t-1}$	0.898***	0.902***
	(0.00)	(0.00)
(ρ_0) Constant	1.069***	1.061***
	(0.02)	(0.02)
N	54230	54230
R^2	0.804	0.803

Table 4 reports the parameter estimates of the AR1 process of user downloads, specified in equation 26. Both OLS and IV estimations show that once a package adopts Python 3, it receives more user downloads over time. The magnitude of the increase depends on r_t, the Python 3 adoption rate in each time period t.

For the same level of user download $x_{i,t-1}$, OLS estimates predict a higher $x_{i,t}$ from Python 3 adoption than IV estimation does. The differences between OLS and IV estimates provide more clues as to endogeneity. In reality, many packages adopt Python 3 even though they have a low level of downloads, probably due to an unobserved positive download shock associated with Python 3, namely, a large $\epsilon^1_{i,t}$.

Table 5 summarizes the estimation results of our model of technology adoption. The estimated discount factor 0.957 is equivalent to a monthly discount factor of 0.993, which is comparable to other papers in the literature (e.g., 0.988 in De Groote and Verboven (2018)). In future versions, we will provide more discussion on the identifiability of the discount factor. The coefficient of α^x, combined with the estimates in Table 4, shows that developers benefit from more user downloads. The negative estimate of AC_0 shows a significant fixed adoption cost. Relative to the fixed cost, incompatible dependencies also serve as significant barriers to adoption. Note that the measure of $\mu_{i,t}$ used in the estimation is the number of incompatible dependencies weighted by the size of packages. The average size of a dependency is 4.89, thus, the average adoption cost due to one incompatible dependency is -1.516, which is

Table 5: Estimated Parameters of Adoption Model

Nonlinear	β	0.957***
Parameters (θ_S)		(0.196)
	α^x	0.690***
		(0.053)
	AC_0	-4.743***
		(0.713)
	α^μ	-0.310***
		(0.052)
	α^{size}	-0.219***
		(0.066)
Log Likelihood		-8019
Number of Packages		4005
Number of Observations		23267

roughly one-third of the estimated fixed cost.

8 Unobserved Heterogeneity

One concern related to the estimation results is sample selection bias due to unobserved heterogeneities. For example, package developers may have a different level of familiarity with Python 3, thus different adoption costs.

The presence of unobserved heterogeneities can bias the estimation results, because the orthogonality assumption between the unobserved components and model inputs is violated. For example, more optimistic developers may care more about the potential long-term benefits, which leads to an overestimation of α^x.

To deal with the unobserved heterogeneities, we employ the method developed by Belzil and Hansen (2002). We assume that there are K types of packages. The probability of belonging to each type $k \in \{1, ..., K\}$ is defined as:

$$p_k = \frac{\exp(q_k)}{\sum_{j=1}^{K} \exp(q_j)} \; and \; q_K = 0.$$

The above equation allows to further rewrite the adoption probability $\hat{p}(\mathcal{S}_{j,t}; \theta)$ in Section

6.5 under the conditional probability:

$$\begin{aligned}\hat{p}\left(\mathcal{S}_{j,t};\theta\right) &\equiv \hat{p}\left(\mathcal{S}_{j,t};\theta,\{q_k\}_{k=1}^{K-1}\right)\\ &= \sum_{k=1}^{K}\hat{p}\left(\mathcal{S}_{j,t}|k;\theta,q_k\right)\times p_k.\end{aligned}$$

Table 6: Estimated Parameters of Adoption Model

Nonlinear	β	0.936***	
Parameters (θ_S)		(0.027)	
	α^x	0.854***	
		(0.221)	with probability
	AC_0 (type 1)	-3.513***	76.9%
		(0.447)	
	AC_0 (type 2)	-7.824***	23.1%
		(1.286)	
	α^μ	-0.328***	
		(0.025)	
	α^{size}	-0.114***	
		(0.042)	
Log Likelihood		-8010	
Number of Packages		4005	
Number of Observations		23267	

The method by Belzil and Hansen (2002) assumes that the type of each individual is implicit and randomly assigned. Table 6 reports the results of the estimation results with the presence of two hidden types ($K = 2$). We assume that there are two types because they naturally represent people who are optimistic about new technology and people who are pessimistic about new technology. We find that allowing for unobserved heterogeneity improves the model estimation, but not significantly. The value of the loglikelihood improves from -8,019 to -8,010.

Model estimation results with $K = 3$ and $K = 4$ are reported in the Appendix. The estimation results do not change with more unobserved types.[25]

[25]We are actively working on unobserved heterogeneity in multiple variables in the model.

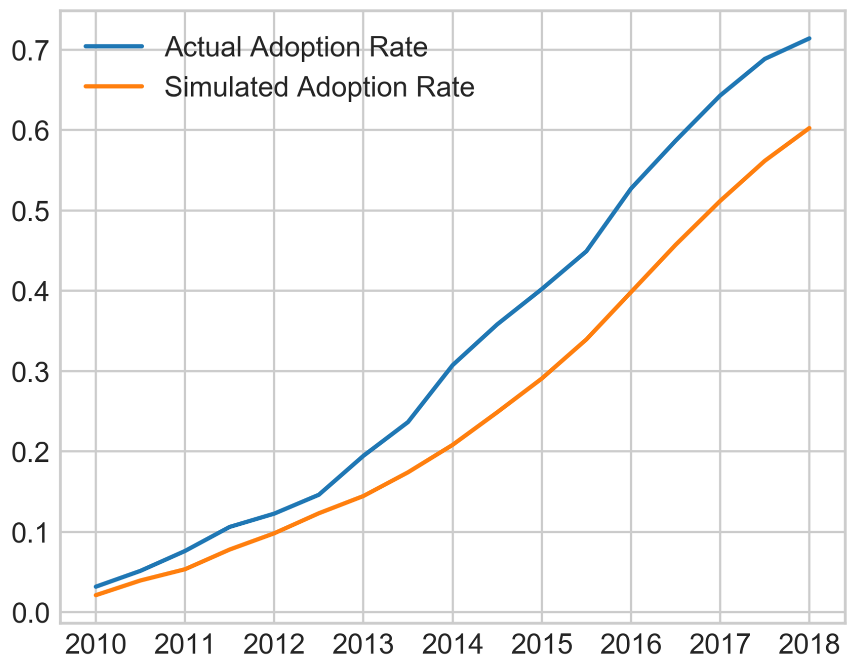


Figure 8: Actual vs. Simulated Adoption Rates

8.1 Model Prediction

One major advantage of the structural model is the ability to run simulations. Through simulation, we can examine the goodness of fit of our model by comparing the actual versus simulated adoption rates over time. It can also be considered as cross validating our model by comparing auxiliary information from the data and the model.

Figure 8 plots the actual adoption rates against simulated adoption rates over time. The adoption rate is calculated as the percentage of packages that have Python 3 support among the sample of 4,005 packages used in the model estimation. The simulation is run using the model without unobserved heterogeneity (Table 5).[26]

Our model predicts a slightly slower Python 3 adoption than the actual adoption decisions. We are currently working on several missing factors that may contribute to the under-prediction: First, unobserved heterogeneity (UH): UH can be incorporated in several variables such as the fixed cost AC_0, valuation of user downloads α^x, etc. Second, selection: early adopters may be fundamentally different from late adopters in many ways. Our dynamic model can be enriched by adding the dimension of changing package types

[26]We are still working on simulation with unobserved heterogeneity.

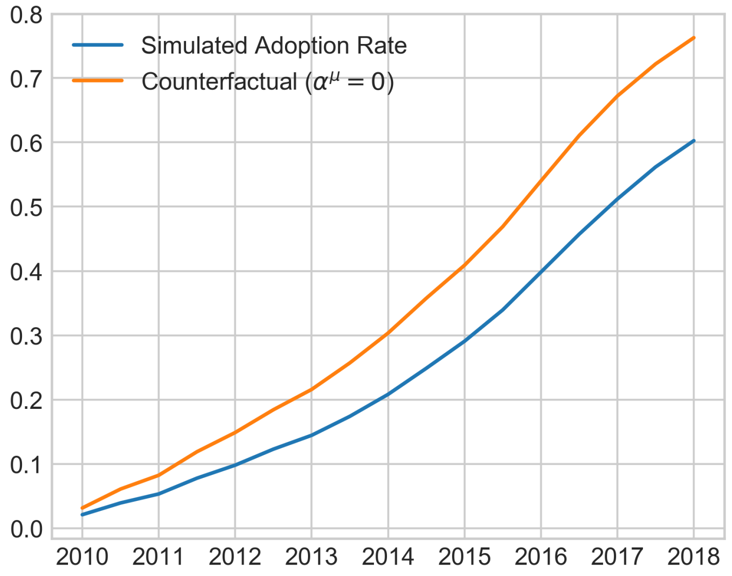


Figure 9: Simulated Adoption Rate Without Network Effects

over time. Third, measurement errors in dependencies: the dependency information is provided by package developers when they upload their package to PyPI. Many packages with dependencies misreport their dependency requirements.

9 Counterfactuals

9.1 Network Effects on Adoption Rate

The model estimation shows that the adoption cost due to one additional incompatible dependency is equivalent to about one-third of the fixed cost. With the structural model, we examine the effect of the network on the speed of adoption through simulation. We simulate the adoption rates from the model, with and without the channel of incompatible dependencies.

Figure 9 contrasts the simulated adoption rates with and without incompatible dependencies. It shows that the adoption cost due to incompatible dependencies contributes to about 1.5 years "inertia" to reach a 60% adoption rate.

9.2 Community-Level Promotion

Katz and Shapiro (1985) predict that the success of a new technology and the speed of technology adoption depends greatly on the "sponsorship". A sponsor is an entity that is willing to make an investment to promote the new technology. The most important sponsor of Python is the Python Software Foundation (PSF). It is a nonprofit organization that oversees various issues in the Python community, including the transition from Python 2 to 3. PSF has been promoting a faster transition to Python 3. However, given the limited amount of effort, it is imperative to understand how the effect can be used most effectively in order to assist the whole industry to switch to Python 3 at a more rapid pace.

One promotion policy to examine is that of promoting Python 3 in different communities. Within the Python community, there are many smaller communities based on the type of audience or domain. Network structure can differ significantly across these communities. Not only can such differences lead to different adoption patterns, they also react differently to counterfactual policies.

The differences in network structure can be seen both within- and across-communities. Packages in certain communities have a more dense or central linkages compared to others; packages in one community are also linked to other packages in a different community. Therefore, a community-level promotion of Python 3 can have a direct effect on that community and an indirect effect on other communities.

Table 7: Description of Python Communities

Community ID	Description
1	text processing & database
2	software development & security
3	website navigation & processing
4	software packing
5	data analysis
6	cryptography & security
7	command line interfaces & remote control
8	content management system
9	website building
10	others

Table 8: Package Characteristics by Community

Community ID	Num. of Packages	Avg Logged Downloads	Avg Age (Years)	Avg Logged Package Size	Avg Num. of Dependencies	Avg Num. of Downstream Packages
1	317	8.442	4.243	3.996	2.389	18.040
2	357	8.687	4.162	3.894	3.393	10.529
3	397	8.150	3.914	3.474	2.605	25.977
4	274	8.702	4.527	3.765	3.623	18.718
5	388	8.154	4.208	5.347	2.785	30.549
6	204	8.786	4.389	4.171	2.827	20.833
7	207	8.280	3.967	3.956	2.727	5.833
8	288	7.877	6.305	4.606	4.751	18.130
9	567	8.326	4.458	3.847	2.407	11.987
10	1006	7.594	3.963	4.050	2.815	6.751

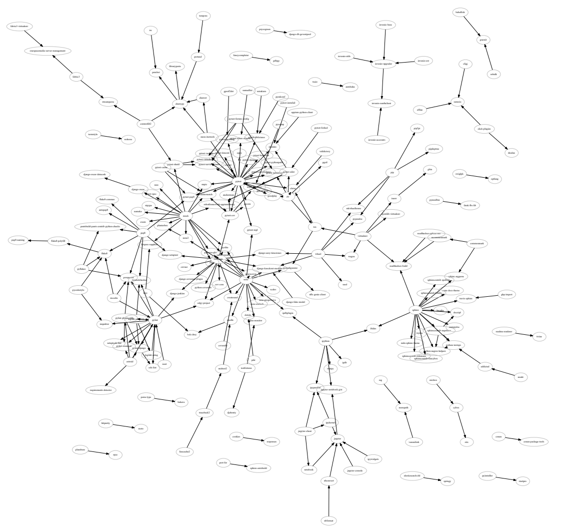
We cluster the 4,005 packages used in our model into nine main communities. Tables 7 and 8 describe the functionalities and characteristics of packages for each community. In the Appendix, Figure 12 plots the network structure for each of the nine communities.

For example, we can see from the picture below, community 3 — Website Navigation (WN) and 5 — Data Analysis (DA) are two leading communities with similar statistics. WN has 397 packages (with 2.60 average dependencies) and DA has 388 packages (with 2.78 average dependencies). Each package in WN has on average 26 downstream packages, and DA has 31. However, these simple statistics cannot capture the detailed network structure.

Figure 10 plots the links for all of the packages in WN and DA. It shows that DA has a more centralized network structure than WN.

We test the effectiveness of community-level promotion in order to promote a faster adoption process. We assume that the promotion can cause package developers in that particular community to change their expectations of the future: the Python 3 adoption rate would increase by 10% permanently. Given the coefficient of r_t value of 0.097, 10% of adoption rate is equal to approximately 0.01 increase in the logrithm of downloads, or about 1% of raw user downloads every period.

The effect of community-level promotion on changes in adoption rate in 2017 is plotted in Figure 11. Communities for each column means the promoted community; communities for each row means the affected ones. For example, promotion activity targeted to community 1


(a) Community 3: Website Navigation (WN)

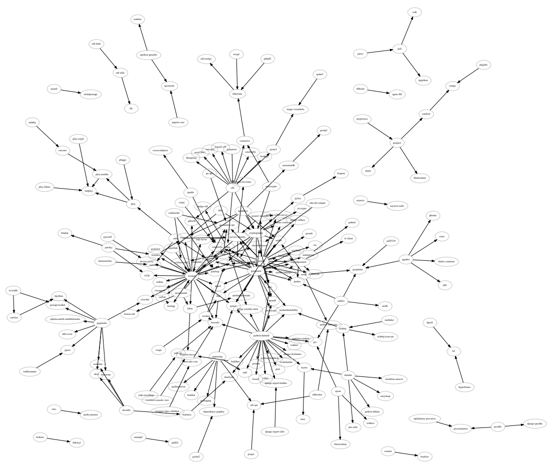

(b) Community 5: Data Analysis (DA)

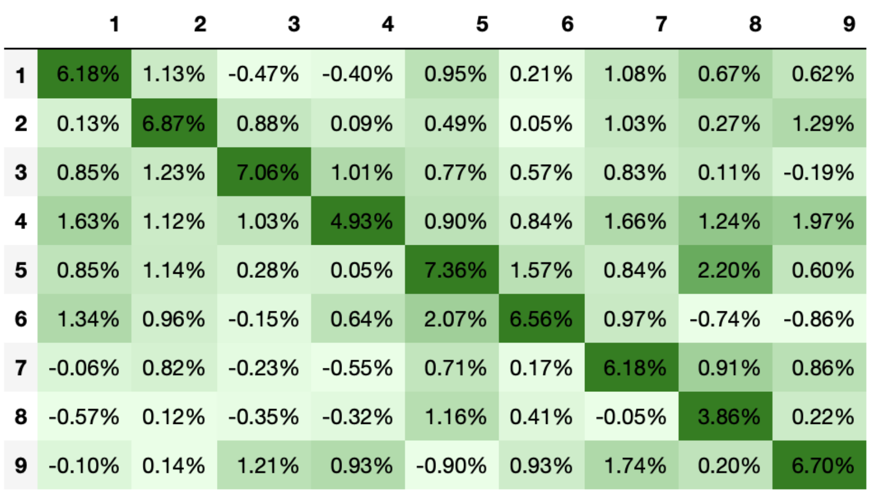
Figure 10: Network Structure of Two Similar Communities

results in a 6.18% increase in adoption rate in community 1, a 0.13% increase in community 2, a 0.85% increase in community 3, etc.

The diagonal items show that the direct effect of promotion within a community can have heterogeneous effects, ranging from 3.86% to 7.06%. Meanwhile, two similar communities with similar direct promotion effects may have very different indirect effects to other related communities.

Take the aforementioned example of community 3 — Website Navigation (WN) and 5 — Data Analysis (DA). Both communities have similar statistics (Table 8) except for the network structure (Figure 10). The direct promotion effect is also similar. However, the indirect effect on other communities differs significantly. A promotion to DA can increase the adoption rate of community 5 — Cryptography & Security (CS) by 2.07%; whereas a promotion to 3 — WN reduces adoption in 5 — CS by 0.15%.

The adverse indirect impact on adoption rate could be due to the following channel: If a package anticipates that its dependencies adopt Python 3 in the near future, then it prefers to delay its own adoption. This channel shows that both the direct and indirect effect of promotion activities should be taken into consideration.



	1	2	3	4	5	6	7	8	9
1	6.18%	1.13%	-0.47%	-0.40%	0.95%	0.21%	1.08%	0.67%	0.62%
2	0.13%	6.87%	0.88%	0.09%	0.49%	0.05%	1.03%	0.27%	1.29%
3	0.85%	1.23%	7.06%	1.01%	0.77%	0.57%	0.83%	0.11%	-0.19%
4	1.63%	1.12%	1.03%	4.93%	0.90%	0.84%	1.66%	1.24%	1.97%
5	0.85%	1.14%	0.28%	0.05%	7.36%	1.57%	0.84%	2.20%	0.60%
6	1.34%	0.96%	-0.15%	0.64%	2.07%	6.56%	0.97%	-0.74%	-0.86%
7	-0.06%	0.82%	-0.23%	-0.55%	0.71%	0.17%	6.18%	0.91%	0.86%
8	-0.57%	0.12%	-0.35%	-0.32%	1.16%	0.41%	-0.05%	3.86%	0.22%
9	-0.10%	0.14%	1.21%	0.93%	-0.90%	0.93%	1.74%	0.20%	6.70%

Figure 11: Changes in Adoption Rate in 2017 with Community Promotion

10 Conclusion

Technological changes have been fundamental to the economic growth, however, many new technologies fail to attract quick and widespread adoption. In many cases, fast adoption of new technologies can be socially beneficial: slow adoption often leads to a long period with incompatible products, while a more rapid adoption enables consumers to better enjoy the convenience brought by the latest technology.

Economic theory has long proposed that the barrier to adopting new technology is due to existing network effects. Our research explores the ways in which technology adoption can be affected by the network structure in a disaggregated input-output network. We find strong evidence that the adoption decisions of downstream players are significantly affected by their upstream counterparts.

This paper also develops a framework by which to measure the impact of such input-output networks on technology adoption. We extend existing dynamic choice models to incorporate a rich input-output network. Taking advantage of the unidirectional property of this network, we build and estimate a dynamic model that allows each agent to anticipate the future actions of others. We believe that this structural framework can be applied to analyze other industries with such an input-output network structure.

References

Abbring, Jaap H and Øystein Daljord. 2018. “Identifying the Discount Factor in Dynamic Discrete Choice Models.” .

Aguirregabiria, Victor and Pedro Mira. 2010. “Dynamic discrete choice structural models: A survey.” *Journal of Econometrics* 156 (1):38–67.

Atkin, David, Azam Chaudhry, Shamyla Chaudry, Amit K Khandelwal, and Eric Verhoogen. 2017. “Organizational Barriers to Technology Adoption: Evidence from Soccer-Ball Producers in Pakistan*.” *The Quarterly Journal of Economics* 132 (3):1101–1164.

Belzil, Christian and Jorgen Hansen. 2002. “Unobserved Ability and the Return to Schooling.” *Econometrica* 70 (5):2075–2091.

Björkegren, Daniel. 2018. “The Adoption of Network Goods: Evidence from the Spread of Mobile Phones in Rwanda.” *The Review of Economic Studies* 37:738–28.

De Groote, Olivier and Frank Verboven. 2018. “Subsidies and Time Discounting in New Technology Adoption: Evidence from Solar Photovoltaic Systems.” *Working Paper* .

DeLong, J Bradford. 2002. “Productivity Growth in the 2000s.” *NBER Macroeconomics Annual* 17:113–145.

Farrell, Joseph and Garth Saloner. 1985. “Standardization, compatibility, and innovation.” *The RAND Journal of Economics* :70–83.

Fershtman. 2016. “Microstructure of Collaboration Network: The Case of Open Source Development.” :1–40.

Fershtman, Chaim and Neil Gandal. 2011. “Direct and indirect knowledge spillovers: the “social network” of open-source projects.” *The RAND Journal of Economics* 42 (1):70–91.

Geroski, P A. 2000. “Models of technology diffusion.” *Research Policy* 29 (4-5):603–625.

Gowrisankaran, Gautam and Marc Rysman. 2012. “Dynamics of Consumer Demand for New Durable Goods.” *Journal of Political Economy* 120 (6):1173–1219.

Gowrisankaran, Gautam and Joanna Stavins. 2004. "Network Externalities and Technology Adoption: Lessons from Electronic Payments." *The RAND Journal of Economics* 35 (2):260–17.

Hall, Bronwyn and Beethika Khan. 2003. "Adoption of New Technology." Tech. rep., National Bureau of Economic Research, Cambridge, MA, Cambridge, MA.

Hall, Bronwyn H. 2009. *Innovation and Diffusion.* Oxford University Press.

Hendel, I and Aviv Nevo. 2006. "Measuring the implications of sales and consumer inventory behavior." *Econometrica* 74 (6):1637–1673.

Katz, Michael L and Carl Shapiro. 1985. "Network externalities, competition, and compatibility." *American Economic Review* 75 (3):424–440.

———. 1986. "Technology Adoption in the Presence of Network Externalities." *Journal of Political Economy* 94 (4):822–841.

Keane, Michael P. 1994. "A Computationally Practical Simulation Estimator for Panel Data." *Econometrica* 62 (1):95–22.

Keane, Michael P and Kenneth I Wolpin. 1997. "The Career Decisions of Young Men." *Journal of Political Economy* 105 (3):473–522.

Magnac, Thierry and David Thesmar. 2002. "Identifying Dynamic Discrete Decision Processes." *Econometrica* 70 (2):801–816.

Mansfield, Edwin, Ruben F Mettler, and David Packard. 1980. "Technology and productivity in the United States." In *The American Economy in Transition.* University of Chicago Press, 563–616.

Rosenberg, Nathan. 1972. "Factors affecting the diffusion of technology." *Explorations in Economic History* 10 (1):3–33.

Rust, John. 1987. "Optimal Replacement of GMC Bus Engines: An Empirical Model of Harold Zurcher." *Econometrica* 55 (5):999–1033.

Rust, John and Christopher Phelan. 1997. "How Social Security and Medicare Affect Retirement Behavior In a World of Incomplete Markets." *Econometrica* 65 (4):781–51.

Ryan, Stephen P and Catherine Tucker. 2011. "Heterogeneity and the dynamics of technology adoption." *Quantitative Marketing and Economics* 10 (1):63–109.

Saloner, Garth and Andrea Shepard. 1995. "Adoption of Technologies with Network Effects: An Empirical Examination of the Adoption of Automated Teller Machines." *The RAND Journal of Economics* 26 (3):479–23.

von Krogh, Georg, Stefan Haefliger, Sebastian Spaeth, and Martin W Wallin. 2012. "Carrots and Rainbows: Motivation and Social Practice in Open Source Software Development." *MIS Quarterly* 36 (2).

Xu, L, T Nian, and Luis MB Cabral. 2016. "What Makes Geeks Tick? A Study of Stack Overflow Careers." *Working Paper, Toulouse School of Economics* .

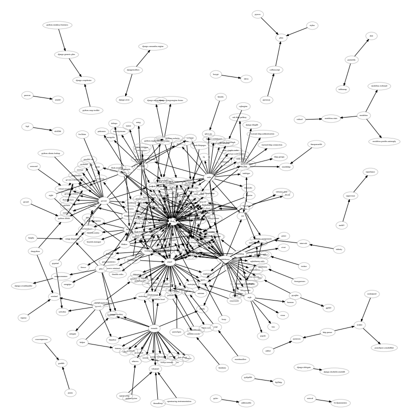
11 Appendix

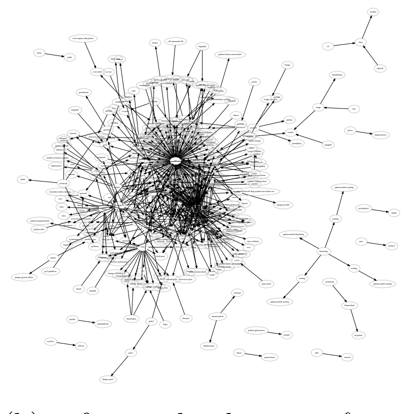
11.1 Examples of Python 3 New/Incompatible Features

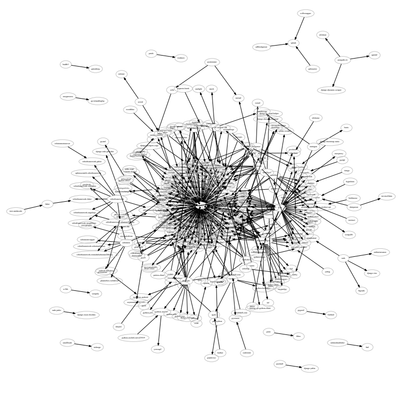
- Default Encoding System
  - Python 2: ASCII
    - e.g. “café” $\longrightarrow$ ’ascii’ codec can’t decode
    - solution: unicode(“cafÃ©”, encoding=’utf8’)
  - Python 3: Unicode
    - e.g. “café” $\longrightarrow$ café
    - unicode(“café”, encoding=’utf8’) $\longrightarrow$ ’unicode’ is not defined
- Division
  - e.g. $f(x) = \text{floor}(\frac{x}{2})$

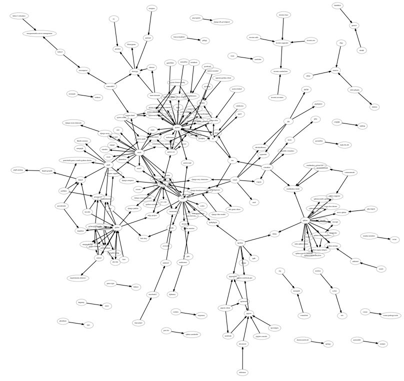
	code	$x = 5$
Python 2	$x/2$	2
Python 2	$x//2$	2
Python 3	$x/2$	2.5
Python 3	$x//2$	2

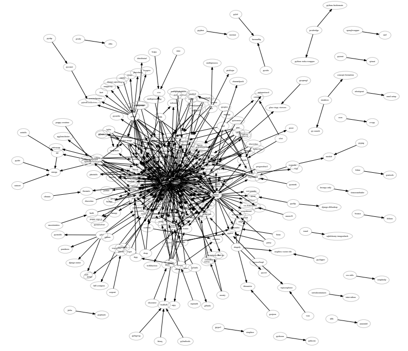
11.2 Network Structure of Python Communities

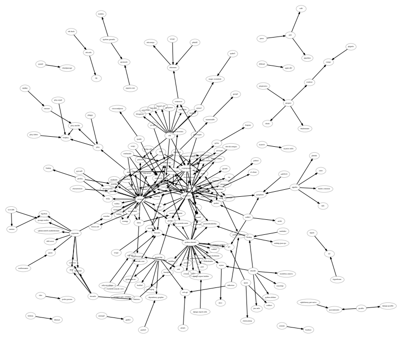

(a) text processing & database

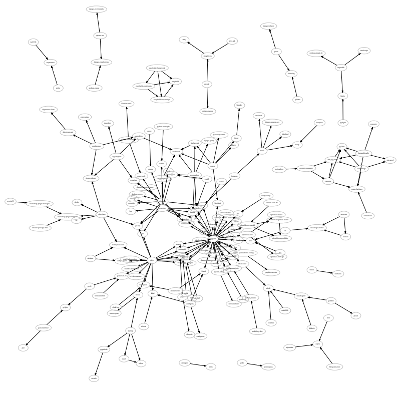

(b) software development & security

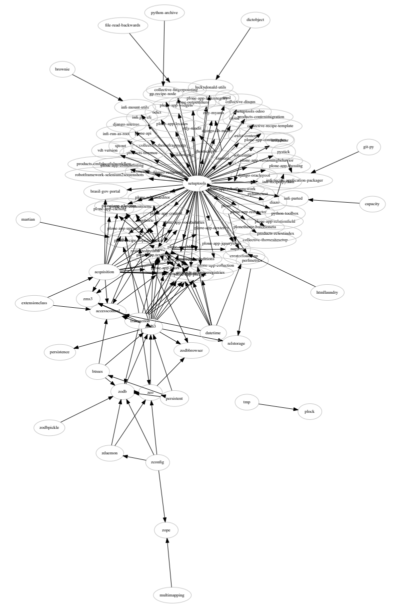

(c) website navigation & processing


(d) software packing


(e) data analysis


(f) cryptography & security


(g) command line interfaces & remote control


(h) content management system

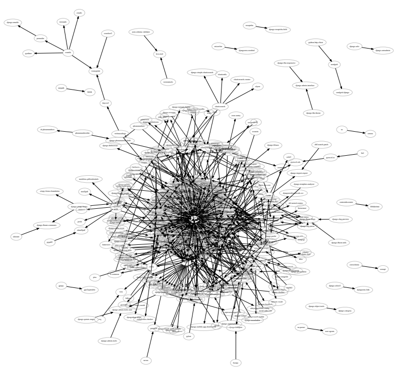

(i) website building

Figure 12: Network Structure of Python Communities