Technology Adoption in Input-Output Networks

Xintong Han and Lei Xu*

February 26, 2020

Abstract

This paper studies how network structure can affect the speed of adoption. In particular, we model the decision to adopt Python 3 by software packages. Python 3 provides advanced features but is *not* backward compatible with Python 2, which implies adoption costs. Moreover, packages form input-output networks through dependency relationships with other packages, and they face an additional adoption cost if the dependency packages lack Python 3 support. We build a dynamic model of technology adoption that incorporates such a network and estimate it using a complete dataset of Python packages. Estimation results show that the average cost of one incompatible dependency is roughly three times the cost for updating one's own code. We conduct counterfactual policies of community-level targeted cost subsidies and show that network structure is crucial to determine an optimal policy of cost subsidy.

*Han: Concordia University and CIREQ, Montreal, Canada. Email: xintong.han@concordia.ca. Xu: Bank of Canada, Ottawa, Canada. Email: lxu@bankofcanada.ca. 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, Gautam Gowrisankaran, Philip Haile, Chun-Yu Ho, Doh-Shin Jeon, Thierry Magnac, Ariel Pakes, Andrew Rhodes, John Rust, Mario Samano, Paul Seabright, Elie Tamer, Robert Ulbricht, Kevin Williams, as well as seminar and conference participants at TSE, Concordia, Edinburgh, Bank of Canada, IE Business School, University of Toronto, and HKBU for helpful comments. Financial support from TSE Digital Center and the NET Institute is gratefully acknowledged. All errors are our own. Views presented in this paper are those of the authors, and do not represent the Bank of Canada's position.

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 on technology adoption shows that many factors can slow the speed of adoptions; 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 demonstrating another channel through which the speed of adoption may be affected—namely, input-output networks.

As the economy becomes more disaggregated, firms specialize in one specific type of product or service and form input-output networks by using the output of other firms as input for their own production.[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, buyers' adoption decisions are also affected and the industry-level 5G adoption may stall (Acemoglu et al. (2012)).[3]

This paper investigates how network structure affects individual technology adoption decisions in a dynamic setting. 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] It provides several fundamental improvements but is *incompatible* with Python 2.[6] In other words, existing code written in Python 2 often fails to work in 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 input-output networks 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 pack-

[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.

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

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

[6]Refer to the Appendix for a list of new functionalities in Python 3 that are incompatible with Python 2.

[7]The two terms "upstream" and "dependency" are used interchangeably throughout the paper.

age, the end user must install the package itself, as well as all of its upstream dependencies.[8]

Our research focuses on the decision to adopt Python 3 made by third-party packages. Deciding to adopt means updating the package to be compatible with Python 3. To use the new features of Python 3, packages need to update their source code. Moreover, another major component of the adoption costs comes from the incompatibility between Python 2 and 3. If any of their upstream dependencies have yet to adopt Python 3, additional development efforts must be made; that is, a higher adoption cost.[9]

Unlike other industries with input-output networks, there is no explicit pecuniary compensation in open-source software (OSS) development and usage. This particular feature allows us to focus on the pure impact of the network structure itself without the complication when price is introduced. We model the utility of OSS as a function of user downloads (Fershtman and Gandal (2011)). For any commonly known motivations behind contributions to OSS (e.g., altruism, career incentives, 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). Our model highlights an intertemporal trade-off: if a package adopts early, it can receive more user downloads over time but pays a higher adoption cost; if a package adopts later, the adoption cost may be lower due to the future adoption decisions of its 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, we draw the input-output structure of packages.[10] We group packages into various layers based on the dependency relationship and calculate the adoption probability layer by layer. Then we propose a novel maximum likelihood estimation (MLE) method to estimate model parameters.

Results from the structural estimation show that upstream dependencies without Python 3 support pose significant barriers in the adoption decisions of downstream packages: the cost of adopting Python 3 with one dependency lacking Python 3 support is equivalent to, on average, three times the cost of updating one's own source code.

The structural model allows us to conduct counterfactual exercises of "sponsorship." A sponsor promotes the new technology, and can affect its future success (Katz and Shapiro (1986)). We evaluate the effectiveness of policies such as community-level targeted cost subsidies to adopt Python 3. We use modularity optimization tools developed in the social network literature to group the packages into various "communities" based on the dependency network: for example, web development and data analysis communities. These communities differ significantly in their network structure. Packages are more densely linked within a

[8]The installation system usually checks whether all dependencies have been installed. If not, it automatically downloads and installs them first. Please see Section 2 for more information.

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

[10]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.

community but less so across different communities. Through counterfactual simulation, we show how technology adoption decisions can propapate through these connected communities. Counterfactual results show that community-level targeted cost subsidies have large heterogeneous effects on adoption rates due to differences in network structure within a community. Moreover, subsidies in one community have significant positive or negative effects on connected communities. We also explore the optimal policy to maximize the effectiveness of cost subsidies to promote a faster Python 3 adoption. The findings imply that policies that consider network structure can significantly improve the effectiveness of promotion.

This paper contributes to several literatures. It contributes to the literature of technology adoption by demonstrating a new channel through which the speed of adoption can be affected. To the best of our knowledge, this paper is the first to measure such adverse effects of input-output networks on technology adoption.

This paper also contributes to the emerging literature that links network analysis and technology adoption. Previous papers on network effects study the topic using a more "reduced-form" approach by modeling utility as a function of the total number of users on the same network.[11] Recent literature start to consider more detailed linkages between individual agents on a network. Ryan and Tucker (2011) measures the heterogeneous network effects from adopting a video-calling technology within a company. Compared to their study, our model accounts for richer patterns of heterogeneity across individual agents. Apart from differences in individual characteristics, we consider the detailed linkages among packages and allow them to discount future utilities differently. By imposing a functional form, we also use several package characteristics that can potentially affect how package developers value future downloads to capture heterogeneity in the discount factor. On the other hand, the literature of social networks has always considered the detailed linkages between individuals, but with limited dynamics. Individuals in reality are inherently dynamic and face intertemporal tradeoffs, and 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)). Björkegren (2018) is one of the first attempts to model technology adoption in a complex network and studies the adoption of mobile phones in Rwanda through the calling network. Björkegren (2018) circumvents the complex dynamic problem by solving per-period equilibria of optimal timing to adopt, and assumes that each individual makes the adoption decision with full knowledge of the actual future adoption time of her contacts. Our approach is based on the dynamic discrete choice model. In this model, individuals decide whether or not to adopt a new technology in the current time period, instead of when to adopt a new technology in a future time period. Agents predict the future adoption probabilities based on the current states. Our approach relaxes the perfect foresight assumption of other agents' fugure adoption decisions, though other simplifying assumptions are needed for the estimation.

[11]Early seminal work includes Farrell and Saloner (1985) and Katz and Shapiro (1986). A more recent overview of the literature can be found in Cabral (2011)

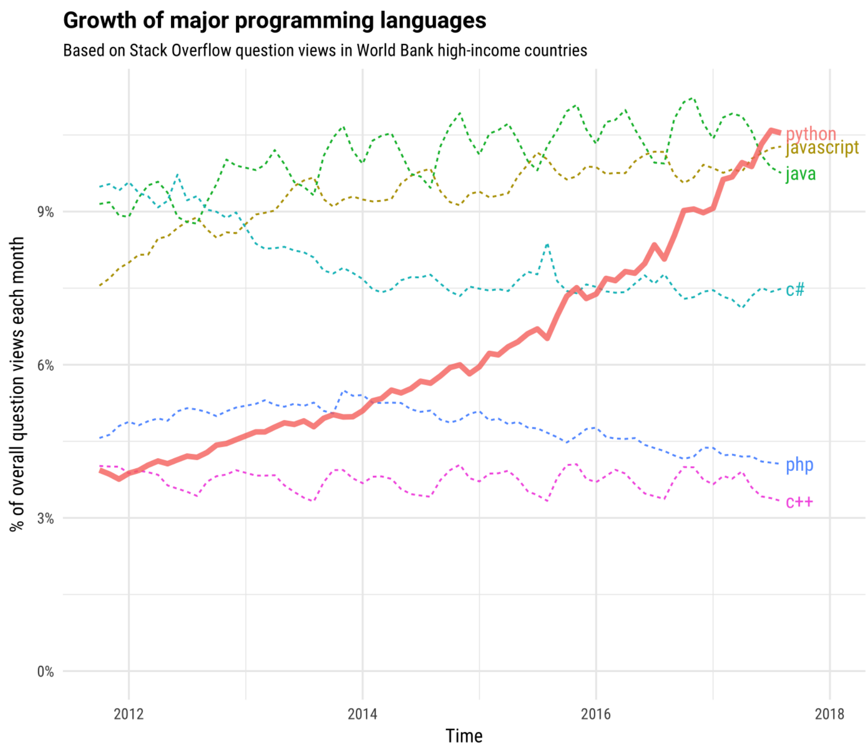


Figure 1: Popularity Trend of Programming Languages

2 Background: The Python Programming Language

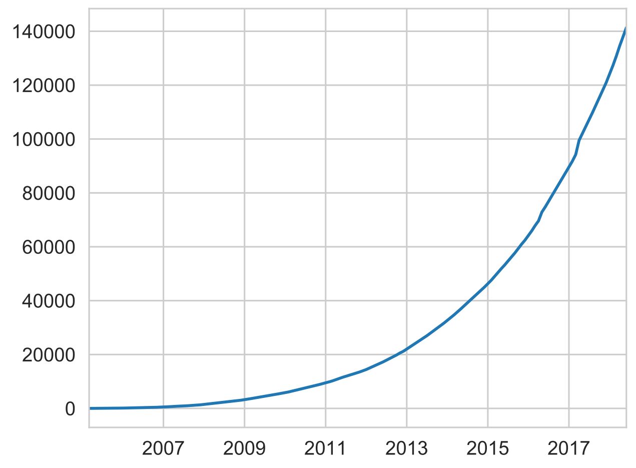
Python is a general-purpose programming language.[12] It has a syntax that allows users to express concepts in fewer lines of code compared with most other major programming languages, and has been widely used in introductory-level computer science courses at universities. The first version was released in 1989 but did not gain popularity until the late 2000s. In the past few years, Python has become one of the most popular programming languages.[13]

Backward compatibility has been a widely disputed topic in the software industry. 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.[14] The trade-off is clear: easier user transition to new technology versus a higher cost of development and slower

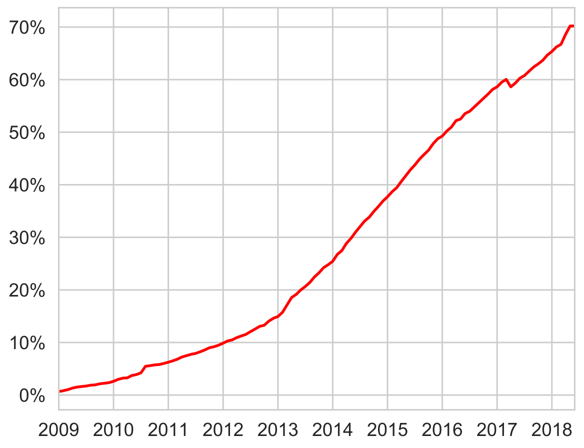
[12]A general-purpose programming language is a computer language that is broadly applicable across application domains. Examples of general-purpose programming languages include C, Java, and Python. It is in contrast to domain-specific language, such as MATLAB (numerical computing), Stata, and R (statistical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

[13]For example, based on the number 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 number one since 2018.

[14]Python core developers are a group of 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 9.1.



(a) Total Number of Packages



(b) % of Packages with Py3 Support

Figure 2: Python Packages with Python 3 Support

innovation. Users and package developers who want to run their code on Python 3 have to update all of the source code to be compatible with Python 3.[15]

Most functionalities on Python are provided by third-party packages. In simple terms, packages can be viewed as a collection of tools and functions that anyone can use to work on more complicated tasks.[16] Packages are also known as libraries, modules, and (sub)routines in other programming languages.

Figure 2(a) plots the number of packages available on Python over time. The exponential growth of packages is another indication that Python itself has gained tremendous popularity over the past decade. Figure 2(b) plots the percentages of packages with Python 3 support after its release in 2008. The transition process has been longer than many expected, and a significant number of packages are reluctant to provide Python 3 support. Those with Python 3 support usually provide two versions for each release: one for Python 2 and another for Python 3. By the end of 2018, roughly 82% of packages support Python 3.

Both the Python programming language and almost all the third-party packages are OSS.[17] The motivations for OSS contributions by software developers can be multifold.[18] The literature of motivations behind private contributions to online public goods suggests that the most common motivations are altruism, career concerns, and ego gratification.

Packages usually specialize in a specific domain and often depend on other packages for related functionalities. For example, *NumPy* is a package that specializes in certain fundamental mathematical operations, such as matrix inversion and multiplication, while

[15]Some packages are developed to help users to transition more easily to Python 3 through automation. However, in most cases, users and package developers still have to test and manually modify much of the code.

[16]A more accurate description and definitions of packages can be found in the official Python documentation at `https://docs.python.org`. Appendix 9.2 also provides a simple example to show how a package is used.

[17]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 require payment for usage.

[18]von Krogh et al. (2012) and Xu, Nian and Cabral (2019) provide overviews of the literature.

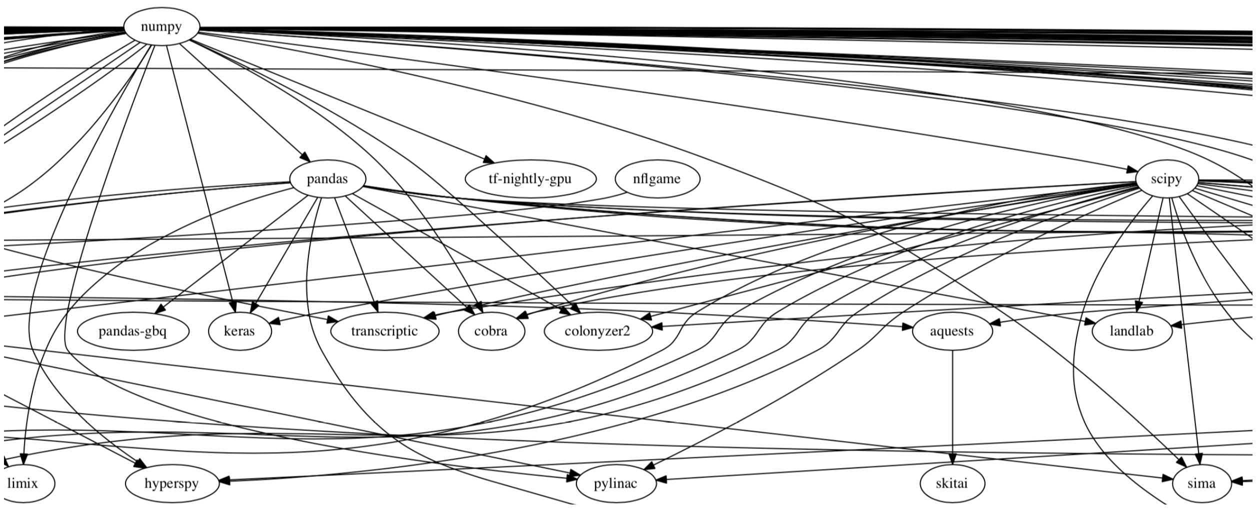


Figure 3: Sample Input-Output Network of Python Packages

SciPy provides more applied tools used in science, such as linear regression, which requires the matrix operations from *NumPy*. In this case, *SciPy* depends on *NumPy*, or *NumPy* is a dependency of *SciPy*. To use *SciPy*, the user has to install both *NumPy* and *SciPy*. When installing a package, the system usually checks whether all the dependencies have been properly installed, and, if not, it automatically downloads and installs the dependency packages first.

Figure 3 shows a small section of the input-output network of Python packages. The arrows represent the dependency relationship.[19] For example, an arrow from *NumPy* to *SciPy* means *NumPy* is the dependency of *SciPy*. In this case, we also refer to *NumPy* as the upstream package to the downstream package *SciPy*.

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 all developers upload their packages so that users can search for and download the packages they need.

When uploading packages to PyPI, package developers usually provide various kinds of information related to the files, such as the description of the package, the contact information for the owners, whether they provide support for Python 2 or Python 3, and what other packages are required as dependencies. [20]

User downloads data consists of two separate sources recorded by PyPI. Before 2016, cumulative download statistics for each file were recorded (Table 2(a)). The system stopped

[19]In principle, the whole input-output network is acyclical. In other words, a circular dependency relationship such as $A \to B \to C \to A$ is not supposed to exist. In the data, there exist a negligible number of cases of circular dependencies (47 out of 90,551 links). We compare the characteristics of all package pairs and manually remove the most "unlikely" link, measured by the number of times a package is used as a dependency.

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

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

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

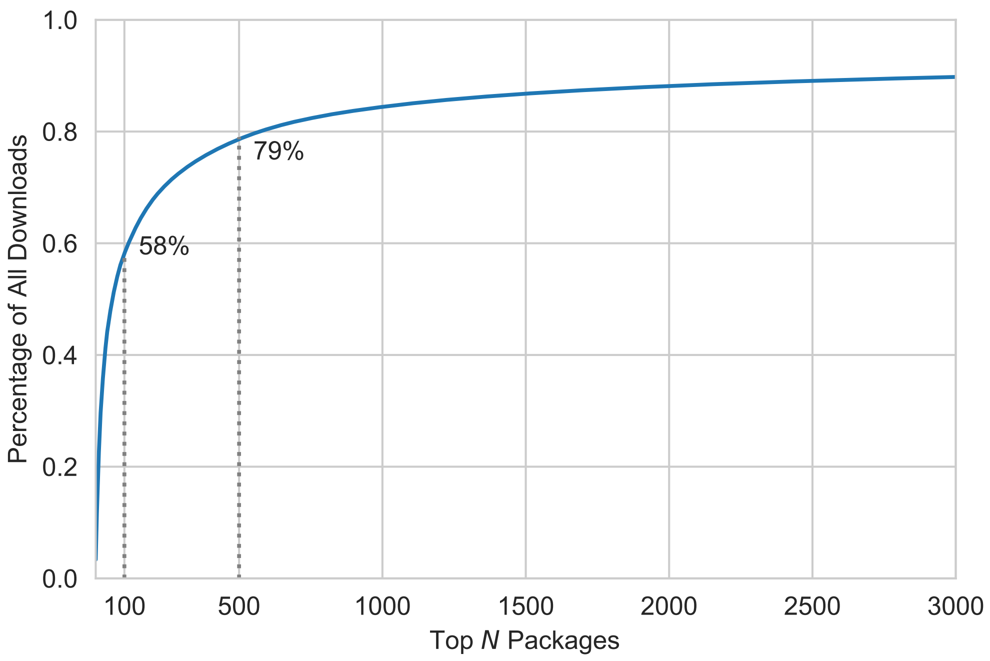


Figure 4: Total Downloads of Top Packages as Percentage of All Downloads

working for a few months from January to May 2016, when PyPI introduced a new system hosted on Google BigQuery. The new system records and publishes certain information related to each individual download (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 2016.[21]

Among the 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. User downloads are concentrated on a relatively few number of popular packages. In 2017, the top 100 packages account for approximately 58% of all downloads, and the top 500 packages account for 79% of all downloads. The long tail is persistent throughout all the years.

We focus on packages that are well maintained, with regular releases by selecting packages based on the following criteria (values in parenthesis are the unconditional percentage of packages that satisfy each measure):

- 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%)

[21]Competing platforms to PyPI also exist (such as Anaconda). All packages on Anaconda are also hosted on PyPI. However, user downloads through Anaconda are not recorded. Throughout our data period, Anaconda holds a relatively small market share. Unfortunately, there is no publicly available information on the downloads statistics of packages hosted on Anaconda.

- Some Python 2/3 Support Information Available (59.9%)

These selection criteria provide 3,397 packages for our analysis.

Table 3: Summary Statistics

	All	Selected	Dep $\geq$ 1
Number of packages	125521	3397	2143
Total downloads (in percentage)	100.00%	46.40%	22.32%
Average logged downloads	9.27	12.34	12.30
per package (min, max)	(3.0, 20.5)	(8.8, 20.5)	(8.8, 19.7)
Average number of dependencies	0.07	2.27	3.60
(min, max)	(0, 53)	(0, 53)	(1, 53)
Average logged package size	4.32	4.42	4.49
(KB) (min, max)	(-2.1, 11.2)	(-2.1, 11.2)	(-0.2, 9.7)

Column "All": all packages available on PyPI at the time of data collection (2018); Column "Selected": selected packages used for estimating the model; Column "Dep $\geq$ 1": those with more than one dependencies among the selected packages.

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 of 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 versus 0.72 in the whole population. These packages are also larger in terms of file sizes. Consistent with other figures in this section, Table 3 implies that our analysis focuses on the well-maintained packages with regular releases that faced an adoption decision of Python 3. These packages are also the most popular packages in the Python community.

4 Model

Our model is based on the single-agent dynamic discrete choice framework developed by Rust (1987) and Keane and Wolpin (1997) to analyze technology adoption decisions by Python packages. Each package i at time t can be described by a state variable $S_{i,t}$. Given the current state $S_{i,t}$, each package i makes an irreversible decision to adopt Python 3, namely, to make the package compatible with Python 3.[22] Let $d_{i,t}$ be a binary irreversible decision:[23]

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

[22]More examples and discussions of irreversible decisions can be found in Rust and Phelan (1997) and Aguirregabiria and Mira (2010).

[23]Given the irreversibility condition, we use $d_{i,t}$ to represent both the adoption decision and status, i.e. once a package adopts Python 3 at time t ($d_{i,t} = 1$), it always supports Python 3 in future periods ($d_{i,t+\tau} = 1 \quad \forall \tau \in N$).

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 (nested) state variables can be specified as the following:

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

The dependency network among packages is incorporated in the last component of the state variable. That is to say, the adoption decision of a package depends on the decisions and states of all of its dependencies as well as the dependencies of dependencies, etc. One implicit assumption embedded in this nested state variable is a sequential move assumption, meaning that package i observes the adoption decision made by its dependencies before making its own adoption decision. Section 4.3 provides more discussion and implications of 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 $S_{i,t}$.

4.1 Value Function and Bellman Equation

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

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

where $u(S_{i,t}, d_{i,t}; \theta)$ is the reward function of the indirect utility, and $C(S_{i,t}; \theta)$ 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:

$$\begin{aligned} & V_\theta(S_{i,t}, d_{i,t-1} = 0) \\ =& \max_{\{d_{i,t+\tau}\}_{\tau=0}^{\infty}} \mathbf{E}_t \Big\{ \sum_{\tau=0}^{\infty} \beta_i^\tau \Big(u(S_{i,t+\tau}, d_{i,t+\tau}; \theta) - D_{i,t+\tau}\, C(S_{i,t+\tau}; \theta) + \nu_{i,t+\tau}^{d_{i,t+\tau}} \Big) | S_{i,t}; \theta \Big\}, \\ & \text{where} \quad D_{i,t+\tau} = \mathbf{1}(d_{i,t+\tau-1} = 0, d_{i,t+\tau} = 1). \end{aligned} \tag{3}$$

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

$$\begin{aligned} & V_\theta(S_{i,t}, d_{i,t-1} = 0) \\ =& \max_{d_{i,t} \in \{0,1\}} u(S_{i,t}, d_{i,t}; \theta) - d_{i,t}\, C(S_{i,t}; \theta) + v_{i,t}^{d_{i,t}} + \beta_i \mathbf{E}_t V_\theta(S_{i,t+1} | S_{i,t}, d_{i,t}) \end{aligned} \tag{4}$$

$$
\begin{aligned}
&= \max\{v_\theta(S_{i,t}, d_{i,t-1}=0, d_{i,t}=0),\ \ v_\theta(S_{i,t}, d_{i,t-1}=0, d_{i,t}=1)\} \\
&= \max\{u(S_{i,t}, d_{i,t}=0;\theta) + v^0_{i,t} + \beta_i \mathbf{E}_t V_\theta(S_{i,t+1}|S_{i,t}, d_{i,t}=0), \\
&\qquad u(S_{i,t}, d_{i,t}=1;\theta) - C(S_{i,t};\theta) + v^1_{i,t} + \beta_i \mathbf{E}_t V_\theta(S_{i,t+1}|S_{i,t}, d_{i,t}=1)\}.
\end{aligned}
$$

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

$$
\begin{aligned}
&V_\theta(S_{i,t}, d_{i,t-1}=1) \\
=&v_\theta(S_{i,t}, d_{i,t-1}=1, d_{i,t}=1) \\
=&\mathbf{E}_t\{\sum_{\tau=0}^{\infty}\beta_i^\tau\Big(u(S_{i,t+\tau}, d_{i,t+\tau}=1;\theta) + \nu^1_{i,t+\tau}\Big)|S_{i,t};\theta\} \\
=&\sum_{\tau=0}^{\infty}\beta_i^\tau\int\Big(u(S_{i,t+\tau}, d_{i,t+\tau}=1;\theta) + \nu^1_{i,t+\tau}\Big)dF_\theta(S_{i,t+1}|S_{i,t}).
\end{aligned}
\tag{5}
$$

We assume that $v^{d_{i,t}}_{i,t}$ are independently and identically distributed according to the type I extreme value distribution. Then the expected value function $\mathbf{E}_t V_\theta(S_{i,t+1}|S_{i,t}, d_{i,t})$ can be calculated using the following equation:

$$
\begin{aligned}
&\mathbf{E}_t V_\theta(S_{i,t+1}|S_{i,t}, d_{i,t}=0) \\
=&\int V_\theta(S_{i,t+1}, d_{i,t}=0)dF_\theta(S_{i,t+1}|S_{i,t}, d_{i,t}=0) \\
=&\int \log\{\sum_{d_{i,t+1}\in\{0,1\}}\exp(v_\theta(S_{i,t+1}, d_{i,t}=0, d_{i,t+1}))\}dF_\theta(S_{i,t+1}|S_{i,t}, d_{i,t}=0)
\end{aligned}
\tag{6}
$$

$$
\begin{aligned}
&\mathbf{E}_t V_\theta(S_{i,t+1}|S_{i,t}, d_{i,t}=1) \\
=&\int V_\theta(S_{i,t+1}, d_{i,t}=1)dF_\theta(S_{i,t+1}|S_{i,t}, d_{i,t}=1) \\
=&\sum_{\tau=1}^{\infty}\beta_i^{\tau-1}\int\Big(u(S_{i,t+\tau}, d_{i,t+\tau}=1;\theta) + \nu^1_{i,t+\tau}\Big)dF_\theta(S_{i,t+1}|S_{i,t}, d_{i,t}=1).
\end{aligned}
\tag{7}
$$

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 $v_\theta(S_{i,t}, d_{i,t-1} = 0, d_{i,t})$. The choice-specific value functions allow us to compute the predicted probability of adopting Python 3 using the standard logit formula:

$$
P(d_{i,t}=1|S_{i,t}, d_{i,t-1}=0;\theta) = \frac{v_\theta(S_{i,t}, d_{i,t-1}=0, d_{i,t}=1)}{\sum_{d'\in\{0,1\}} v_\theta(S_{i,t}, d_{i,t-1}=0, d')} \tag{8}
$$

$$
P(d_{i,t}=0|S_{i,t}, d_{i,t-1}=0;\theta) = 1 - P(d_{i,t}=0|S_{i,t}, d_{i,t-1}=0;\theta) \tag{9}
$$

$$
P(d_{i,t}=1|S_{i,t}, d_{i,t-1}=1;\theta) = 1 \tag{10}
$$

$$
P(d_{i,t}=0|S_{i,t}, d_{i,t-1}=1;\theta) = 0. \tag{11}
$$

4.2 Payoff Function

As discussed in Section 2, for any of the most commonly known motivations to contribute to the OSS development, contributors should prefer having more user downloads. Therefore, following the literature of OSS (Fershtman and Gandal (2011) and Fershtman and Gandal (2008)), we model the payoff of package developers as a function of user downloads. Denote $x_{i,t}$ as the logarithm of the 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:

$$u(S_{i,t}, d_{i,t}; \theta) = \alpha^x \, x_{i,t}(x_{i,t-1}, d_{i,t}; \theta). \tag{12}$$

Furthermore, we assume that the evolution of the demand follows a first-order Markov process:

$$\begin{aligned} x_{i,t} = \rho_0 + \rho_1 \cdot x_{i,t-1} + \rho_3 \cdot ds_{i,t} \\ + (\rho_2 \cdot x_{i,t-1} + \rho_4 \cdot ds_{i,t} + \rho_r \cdot r_t) \cdot d_{i,t} + \epsilon_{i,t}, \end{aligned} \tag{13}$$

where d indicates a package i's adoption status/decision $d_{i,t}$; $ds_{i,t}$ is the number of packages that specify i as their dependency package, that is, the number of i's downstream packages; r_t is the current Python 3 adoption rate among all packages; and $\epsilon^0_{i,t}, \epsilon^1_{i,t}$ are two white noises that are normally and interdependently distributed with mean 0 and variance σ^2_ϵ.[24]

In particular, in order to make our model more general, our model tries to capture the heterogeneities of adoption benefit in two dimensions: the timing of adoption and the characteristics of a package. From equation 13, the additional benefit of package i adopting at time t is:

$$\frac{\partial}{\partial d_{it}} x_{it} = \rho_2 \cdot x_{i,t-1} + \rho_4 \cdot ds_{i,t} + \rho_r \cdot r_t.$$

The first two components capture heterogeneity in adoption benefits due to the popularity and importance of a package. A package with a large existing Python 2 user base is also more likely to have a significant additional number of potential users on Python 3. The adoption by an important package with many downstream packages may cause a cascade effect for many others, and such a package itself also benefits from the direct downloads from it. The third component captures the timing of adoption. When Python 3 is not widely adopted, the potential Python 3 user base is small; thus, a package doesn't gain many downloads from adopting Python 3.

We model user demand with a parsimonious first-order Markov process instead of a structural model of user adoption, mostly due to data limitations. We only observe the total number of downloads for each file of a package, which is not equivalent to the user base because a user can download a package multiple times. More importantly, without user identifiers such as IP addresses, there is no way to identify the Python adoption decision for each individual. We assume symmetric information in terms of user downloads because package developers have no more download statistics than econometricians. Lastly, the parsimonious first-order Markov process can well capture the variations of downloads in the

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

data (see Section 6).

4.3 Adoption Cost

We model the cost function as a function of an agent's decision $d_{i,t}$, lines of code $s_{i,t}$, and the number of incompatible dependencies $\mu_{i,t}$ (adjusted by an "importance" measure of that dependency). The switching cost is defined as below:

$$C_{i,t} = c_0 + \alpha^{\mu}\,\mu_{i,t} + \alpha^{s}\,s_{i,t}, \tag{14}$$

and we further assume that $\mu_{i,t}$ is specified as the following:

$$\mu_{i,t} = \sum_{j \in U_i} \mathbf{1}\{d_{j,t} = 0\}\,s_{j,t}, \tag{15}$$

meaning the number of incompatible dependencies is weighted by package sizes. We prefer this to the raw number of dependencies because, in reality, the cost to deal with an incompatible dependency may be heterogeneous. For example, it may be easier to find a dependency alternative for another small dependency compared to a large one.

The fixed component c_0 of the adoption cost captures all other costs not captured by $\mu_{i,t}$ and $s_{i,t}$. One important component is the difference between Python 2 and 3, or the difficulty level for a developer to learn Python 3. Since it is a one-time cost, c_0 also includes the present value of future maintenance costs.

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

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 is more appropriate to model a sequential-move game; if not, then a simultaneous-move game would be more appropriate.

We prefer the sequential-move assumption instead of a simultaneous one for the following reasons. First, upstream packages tend to be the more popular ones that often pre-announce their plans for future releases, including the Python 3 adoption decision. 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 that 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 will be specified in detail in Section 5.1.

We do not think that a simultaneous-move assumption and a sequential one would produce significantly different results. The only observations that cause 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 in the data.[25]

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

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 first-order Markov process of user downloads, which includes both package and network characteristics.[26] Assumption 2 implies that a package does not explicitly include the response 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. In other words, it is likely that indirect downloads through downstream packages account for a small portion of total user downloads.[27] Moreover, an upstream package often has many downstream packages and is unlikely to track all the downstream packages dependent on it.

4.4 State Variables and 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, 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 the decisions and states of its dependencies $\{d_{j,t}, S_{j,t}\}_{j \in U_{i,t}}$ insofar as it cares about its own adoption cost (a function of $\mu_{i,t}$), as well as its future evolution given the current states. The law of motion of $\mu_{i,t}$ can be expressed in the following way:

$$\begin{aligned} & \mathbf{P}(\mu_{i,t+1} = \mu' | \mu_{i,t} = \mu, \{d_{j,t}, S_{j,t}\}_{j \in U_{i,t}}; \theta) \\ = \; & \mathbf{P}(\mu' | \mu, \{d_{j,t}, S_{j,t}\}_{j \in U_{i,t}}; \theta), \end{aligned} \tag{16}$$

where $\mu \geq \mu'$, meaning that the adoption cost in future periods might be lower. The component $\{d_{j,t}, S_{j,t}\}_{j \in U_{i,t}}$ is important to predict the future adoption probabilities of dependency j. Denote the probability of dependency j adopting and not adopting Python

[25]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.

[26]For example, a package might receive much of its downloads indirectly through a popular downstream package, thus having stronger incentive to adopt Python 3 earlier due to the fear of being dropped as a dependency. Such strategic interactions are not explicitly modeled, but the effects are implicitly approximated in the first-order Markov process.

[27]Unfortunately, neither the econometrician nor the packages have a clear knowledge regarding the portion of downloads coming from direct versus indirect channels. The data limitation is another reason why we make such an assumption.

3 as $\widehat{p}^1_{j,t+1} = \mathbf{P}(d_{j,t+1} = 1|d_{j,t} = 0, S_{j,t};\theta)$ and $\widehat{p}^0_{j,t+1} = \mathbf{P}(d_{j,t+1} = 0|d_{j,t} = 0, S_{j,t};\theta)$, respectively. Further, we define the set of dependencies without Python 3 support as $\Omega_{i,t} \equiv \{j \in U_{i,t}, d_{j,t} = 0\}$. Then we can simplify equation 16 as:

$$\mathbf{P}(\mu'|\mu, \{\widehat{p}_{j,t+1}\}_{j\in\Omega_{i,t}};\theta). \tag{17}$$

Note that μ is defined as the number of incompatible dependencies weighted by the package size. It is a deterministic function of the set of incompatible dependencies. Thus the transition probability of μ is equivalent to that of the set of incompatible dependencies. Define $\mathcal{O}_{i,t}$ as the power set of $\Omega_{i,t}$ that contains all possible subsets of $\Omega_{i,t}$, then the equations above can be further simplified as:

$$\begin{aligned} &\mathbf{P}(o' \mid \Omega_{i,t}, \{\widehat{p}_{j,t+1}\}_{j\in\Omega_{i,t}};\theta) \\ = &\prod_{j\in o'} \widehat{p}^0_{j,t+1} \prod_{j\in\Omega_{i,t}\backslash o'} \widehat{p}^1_{j,t+1}, \end{aligned} \tag{18}$$

where $o' \in \mathcal{O}_{i,t}$, $\mu = \sum_{j\in\Omega i,t} s_{j,t}$, and $\mu' = \sum_{j\in o'} s_{j,t}$.

4.5 Transition Matrix

The calculation of EV in equation 7 depends crucially on the specification of the transition matrix, or $\mathbf{P}_{S'|S}$.

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 incompatible 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 simple. We assume that it follows a first-order Markov process specified in equation 13, with the parameter values estimated outside of the value function iteration. In contrast, the law of motion of $\mu_{i,t}$ is much more difficult.

One of the most important trade-offs for package i to adopt Python 3 today at t versus 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 $t + 1$:

$$\widehat{p}^1_{j,t+1} = \int_{S_{j,t+1}} \mathbf{P}(d_{j,t+1} = 1|S_{j,t+1}, d_{j,t} = 0, z_j;\theta)d\mathbf{P}(S_{j,t+1}|S_{j,t}, d_{j,t} = 0, z_j;\theta). \tag{19}$$

The integral can then be computed through a simulation of future states $S_{j,t+1}$. Let $S^m_{j,t+1}$

be the mth simulation, then the value of $\widehat{p_{j,t+1}^1}$ can be obtained by:

$$\widehat{p}_{j,t+1}^1 = \frac{1}{M}\sum_{m=1}^{M}\mathbf{P}(d_{j,t+1} = 1|S_{j,t+1}^m, d_{j,t} = 0, z_j; \theta). \tag{20}$$

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 dependencies, 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 Section 5, 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 $\widehat{p}_{j,t+\tau}^1$ for each of $j \in \Omega_{i,t}$ and $\tau \in \mathbb{N}$.

Given the transition probability specified in equation 18, the full transition matrix needed to calculate $EV(S, d = 0; \theta)$ can be specified as the following:

$$P(o' \in O_{i,t}|o \in O_{i,t}) = \begin{cases} 0 & \text{if } o \nsubseteq o' \\ \prod_{j\in o'}\widehat{p}_{j,t}^0 \prod_{j\in o\backslash o'}\widehat{p}_{j,t}^1 & \text{if } o \subseteq o' \end{cases}. \tag{21}$$

The calculation of the transition matrix, as specified in equation 21, can be illustrated using the same example in Section 4.4. 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 $\widehat{p}_{A,t+1}^1 = a$ and $\widehat{p}_{B,t+1}^1 = b$. The powerset of $\Omega_{i,t}$ is $\mathcal{O}_{i,t} = \{\varnothing, \{A\}, \{B\}, \{A, B\}\}$, and $\mathcal{O}_{i,t}^0 = \{\varnothing\}$. Therefore, the transition matrix of μ can be calculated using equation 21:

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

The corresponding value of μ for each row (or column) is $s_A + s_B, s_A, s_B, 0$, 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. Dependency packages A and B differ not only in their sizes that result in different values of $\mu_{i,t}$ but also in other aspects (such as downloads) that affect their own future adoption probabilities that matter for the optional value of waiting for package i. The transition matrix calculated using equation 21 considers all such cases.

4.6 Discount Factor

To capture the heterogeneity in valuation of future utility, we specify the discount factor β_i as a form of Gumbel function $\mathcal{B}(z_i' \lambda)$:

$$\beta_i \equiv \mathcal{B}(z_i' \lambda) = 1 - e^{-e^{z_i' \lambda}} \tag{22}$$

$$\text{where } z_i' \lambda = \lambda_0 + \lambda_1 \, Releases_i + \lambda_2 \, Files_i + \lambda_3 \, Description_i + \lambda_4 \, Classifiers_i. \tag{23}$$

The Gumbel specification has an important advantage in numerical estimation because, for any value of the argument, $z_i' \lambda$, $\mathcal{B}(z_i' \lambda)$ is by definition bounded within 0 and 1, and no further constraints are needed. We further specify $z_i' \lambda$ as a linear function of several variables that can potentially affect how package developers value future downloads but not its Python 3 adoption decisions: $Releases_i$: average number of releases per year; $Files_i$: average number of files contained in each release;[28] $Description_i$: average number of characters used in the description section; and $Classifiers_i$: average number of classifiers (similar to tags or labels) associated with each package.[29]

5 Identification and Estimation

The parameters of interest in our model are the following:

$$\theta = \{\underbrace{\rho_0, \rho_1, \rho_2, \rho_3, \rho_4, \rho_r}_{\theta_D}; \ \underbrace{c_0, \alpha^\mu, \alpha^s, \lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4}_{\theta_S}\}.$$

The parameters can be grouped into two subsets: θ_D and θ_S. θ_D includes parameters of the demand function, the process of user downloads modeled as a first-order Markov process; θ_S includes parameters of the supply side, which is the structural model of technology adoption by package developers.

5.1 Identification

The set of demand-side parameters $\theta_D = \{\rho_0, \rho_1, \rho_2, \rho_3, \rho_4, \rho_r\}$ 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 the Markov process separately from the structural model of technology adoption for the reason mentioned in 4.2.

[28]For each release of a package, there can be multiple files for different operating systems (Windows, macOS, Linux, etc.) and for different Python versions.

[29]Some examples of "Classifiers" include: Environment - Web Environment, Intended Audience - Science/Research, License - OSI Approved - MIT License, Topic - Internet - WWW/HTTP.

One major concern is that the estimates of the AR1 process may suffer from an endogeneity problem. The AR1 process is estimated using the evolution of user downloads as 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. To correct the endogeneity problem, we instrument the adoption decision $d_{i,t}$ using package characteristics such as the number of downloads and package size.[30]

Then the estimates of the AR1 process of x are used as inputs for the structural model of technology adoption. The fixed cost c_0, the cost due to the network α^{μ}, and the cost due to package size α^{s} are identified through the variations of x, μ, s, and the adoption decisions d.

In most settings of dynamic discrete choice models, the discount factor is not separately identified from other parameters (Magnac and Thesmar (2002)). In our setting, all utilities are measured in terms of user downloads. As a result, α^{x} and β_i play a similar role in the model. The identification of the discount factor requires certain variations that shift future but not current utility (Abbring and Daljord (2019)). In theory, such variations can be found in our data: imagine a case where two packages, A and B, that are identical in every way except for their dependencies are the same size. In this case, A and B also face identical adoption costs with the same μ. However, due to differences in other characteristics of their dependencies (such as downloads), the dependencies can have different future adoption probabilities, which in turn affect the optional value of waiting for A and B. In practice, however, there are very few cases, and the identification power from this source is rather weak.[31]

By fixing the value of α^{x}, we can identify the discount factor through the differences in package-specific characteristics, which affect package developers' intertemporal trade-offs between short-term adoption cost and long-term benefit, which is exogenously given from the demand side. For example, two packages face identical adoption costs but have a slight difference in user downloads. Assume that the initial differences in downloads also lead to persistent long-term differences. If such small differences lead to large discrepancies in their adoption probability, it means that packages are more patient and vice versa.

In this paper, we fix the value of α^{x} to 1. Note that the demand side of user downloads is estimated separately. Fixing α^{x} means fixing the current period's utility per unit of download. With that, we allow β to vary with certain package characteristics that are unlikely to affect the adoption decisions. The discount-factor-related parameters λs are identified through the variations of such variables and the observed adoption decisions. For example, a package with many "classifiers" is likely to put more weight on the additional downloads as a result of Python 3 adoption.

[30]As a robustness check, we also conducted 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 instrument 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. Steps 1 and 2 are repeated until all of the estimates converge to a fixed point.

[31]That is to say, the nonlinear optimizer still returns reasonable estimates but the objective function is nearly flat near the solution.

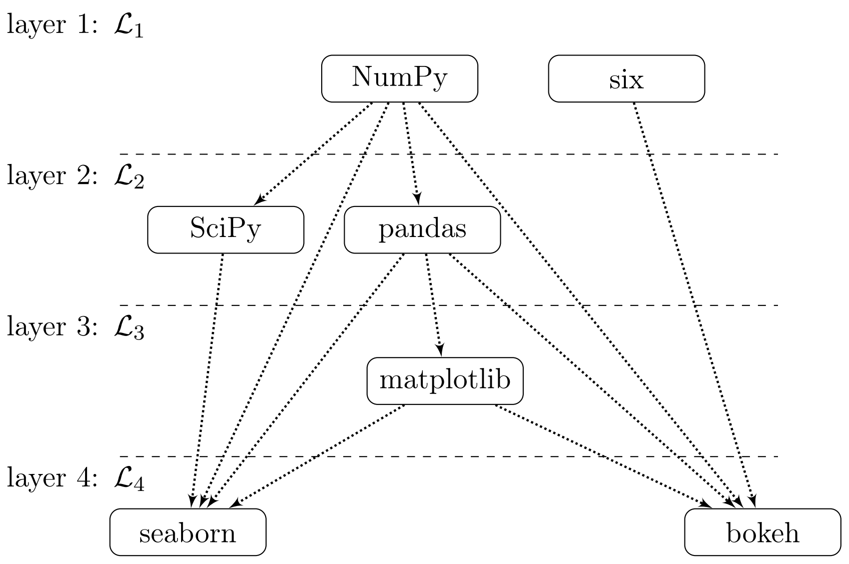


Figure 5: Example of the Layered Network Representation

5.2 Estimation Method

Our model of technology adoption is estimated using the MLE method. The likelihood function is defined as:

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

where $\widehat{p}^1_{i,t} \equiv \widehat{p}^1(S_{i,t};\theta)$ as defined in equation 8.

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 and calculate Python 3 adoption probability for packages sequentially.[32] To do this, we first group all packages into L sets based on their acyclical dependency relationship: $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \cdots, \mathcal{L}_L$ where $\mathcal{L}_l$ is defined as:

$$\mathcal{L}_l = \{i | \max_j n_{ji} = l-1 \quad \forall j \in N\},$$

where N is the set of all packages and n_{ji} is the minimal length of all directed acyclical graph from package j to i. Figure 5 depicts layered network representation using a small sample of Python packages.

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, ..., etc. The probability of adopting Python 3 for each of the

[32]Assumption 1 simplifies our model estimation, but only slightly. 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. Such cases comprise a small percentage of cases in the data.

upstream packages is then used for the decision-making process of the downstream package, in order to calculate the transition probability of $\mu_{i,t}$.

Regarding the timing, we summarize all the variables to six-month periods to fit our dynamic discrete-choice model. For example, 2013/01/01 to 2013/06/30 is one period, and 2013/07/01 to 2013/12/31 is another.

Our algorithm begins by setting initial values for θ_D, which come from the estimation of the AR1 process of user downloads, specified in equation 13. Estimation then involves iteration on the four steps, where the m^{th} iteration follows.

Step 1: Estimate Demand Parameters θ_D Using IV

- Estimate the first-order Markov process of demand function using $x_{i,t}, \mu_{i,t}, s_{i,t}$ as IV with the standard IV estimation method.

Step 2: Estimation of the Model of Technology Adoption

- Given the estimates of demand function θ_D and an initial guess of θ_S:
  - for $i \in \mathcal{L}_1$, build transition matrix for each i, t and calculate $\widehat{p}^1(S_{i,t}; \theta)$
  - for $i \in \mathcal{L}_2$, given $\widehat{p}^1(S_{j,t}; \theta)$ $\forall j \in \mathcal{L}_1$, build transition matrix for each i, t and calculate $\widehat{p}^1(S_{i,t}; \theta)$
  - for $i \in \mathcal{L}_3$, given $\widehat{p}^1(S_{j,t}; \theta)$ $\forall j \in \mathcal{L}_{1,2}$, build transition matrix for each i, t and calculate $\widehat{p}^1(S_{i,t}; \theta)$

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

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

6 Results

Table 4 summarizes the parameter estimates of the download process, modeled as a first-order Markov process specified in equation 13. Column 1 shows results from OLS regression and column 2 corrects the endogeneity of $d_{i,t}$ as explained in Section 5.

Both OLS and IV estimation results indicate heterogeneous effects on downloads after adopting Python 3. When Python 3 is not yet widely adopted (low r_t), a package with few downstream packages (low $ds_{i,t}$) may find that Python 3 adoption negatively affects the number of downloads. It is likely to be due to the time constraint faced by package

Table 4: Estimation of Download Process (First-Order Markov)

	(1)	(2)
	OLS	IV
$x_{i,t-1}$	0.779***	0.788***
	(0.00)	(0.00)
$ds_{i,t}$	0.167***	0.167***
	(0.00)	(0.01)
$d_{i,t} \times x_{i,t-1}$	-0.002	-0.014***
	(0.00)	(0.00)
$d_{i,t} \times ds_{i,t}$	0.080***	0.086***
	(0.01)	(0.01)
$d_{i,t} \times r_t$	0.216***	0.314***
	(0.03)	(0.03)
Constant	2.001***	1.958***
	(0.02)	(0.02)
N	46266	46266
R^2	0.831	0.831

developers: more time spent to adopt Python 3 means less time to make improvements to the existing Python 2 version of the package. However, the incentive for packages with more downstream packages can be much larger. Once such a package adopts Python 3, it clears an important roadblock for its downstream packages and allows them to adopt Python 3 with less cost. The upstream package itself also benefits because it gets both direct downloads from end users and indirect downloads through the installation of its downstream packages. The benefits of such a cascade effect are captured through $ds_{i,t}$ in this downloads process.

Table 5 summarizes the estimation results of the structural model of technology adoption. It lists the parameter estimates of the cost function and the discount factor in separate parts.

The cost function of adopting Python 3 has three components: 1. fixed adoption cost c_0; 2. cost to deal with dependencies without Python 3 support $\alpha^\mu \, \mu_{i,t}$; and 3. cost to update one's own code $\alpha^s \, s_{i,t}$. Both $\mu_{i,t}$ and $s_{i,t}$ are measured in the unit of logarithm of file sizes, and packages differ significantly in their sizes. To contrast the magnitude of the cost components, we plot the distribution of costs due to one incompatible dependency and updating one's own package in Figure 6. The average logarithm of package size for a dependency is 5.05, and the average cost of dealing with one incompatible dependency is $\alpha^\mu \times 5.05 = 1.21$. In comparison, the average size for a package is 4.52, and the average cost of updating one's code is $\alpha^s \times 4.52 = 0.46$, which is roughly one-third of the cost of dealing with one incompatible dependency. The relative high cost of dealing with one incompatible dependency is not too surprising. Package developers are more familiar with their own code and less so with the functionalities provided by a dependency package; thus, manually updating part of the code from a dependency for one's own use can be challenging. At the same time, it is not always easy to find a good alternative package that provides exactly the same functionality as the

Table 5: Parameter Estimates of Adoption Model (θ_S)

Cost Parameters	c_0	-1.741***
		(-19.741)
	α^μ	-0.24***
		(-13.78)
	α^s	-0.103***
		(-5.153)
Discount Factor Parameters	λ_0	-0.736***
		(-3.055)
	λ_1	0.18***
		(6.021)
	λ_2	-0.412***
		(-3.565)
	λ_3	0.03
		(0.638)
	λ_4	0.258***
		(4.931)
Log Likelihood		-6749
Number of Packages		3397

existing dependency.[33]

The rich specification of the discount factor in equation 23 allows us to examine the heterogeneity in the way different packages value future changes in cost and benefits.

The mean value of β_i for the 3,397 packages used in our analysis is 0.697 (for six months), which is equivalent to an annual discount factor of 0.486. This estimate is much lower compared to the discount factors estimated for other industries (e.g., 0.988 in De Groote and Verboven (2019)). This is not too surprising because OSS differs importantly from other industries due to a lack of explicit pecuniary incentives. To examine the heterogeneity in β_i in detail, we plot the distribution of $\hat{\beta}_i$ in Figure 7. It displays a large dispersion ranging from 0.326 to 0.944 with a mode of 0.73.

6.1 Model Fit

One 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.

We run simulations of adoption decisions. Based on the states of each time period, we simulate the Python 3 adoption decisions for packages that are still active in that period. We run the same simulation 100 times and plot the average simulated adoption rate against the

[33]In theory, the cost to deal with an incompatible dependency depends on the availability of alternative packages. Unfortunately, we are unable to account for this factor in the cost function due to the lack of a systematic method to identify competing packages.

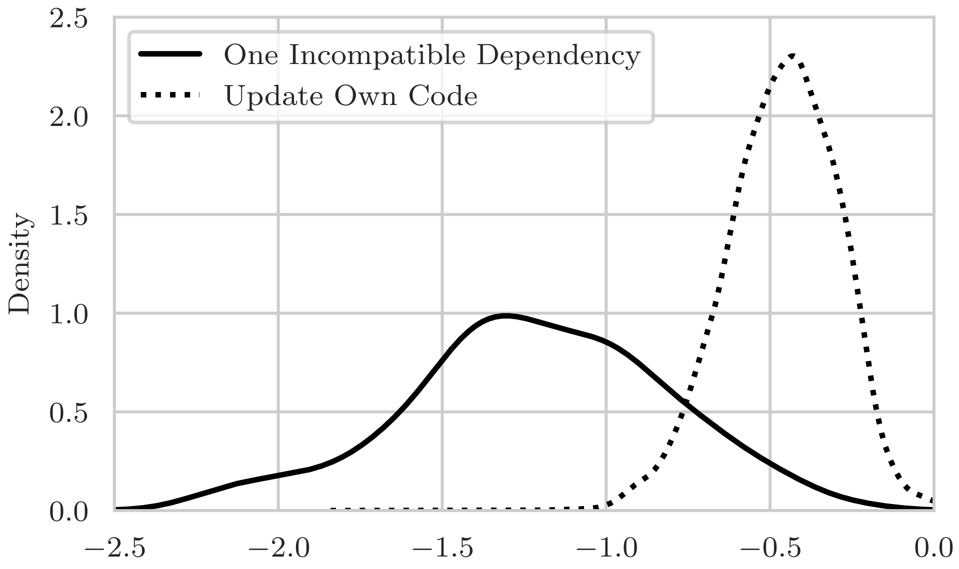


Figure 6: Variable Cost Due to One Incompatible Dependency Versus One's Own Code (convert α^{μ}, α^{s} to the same scale)

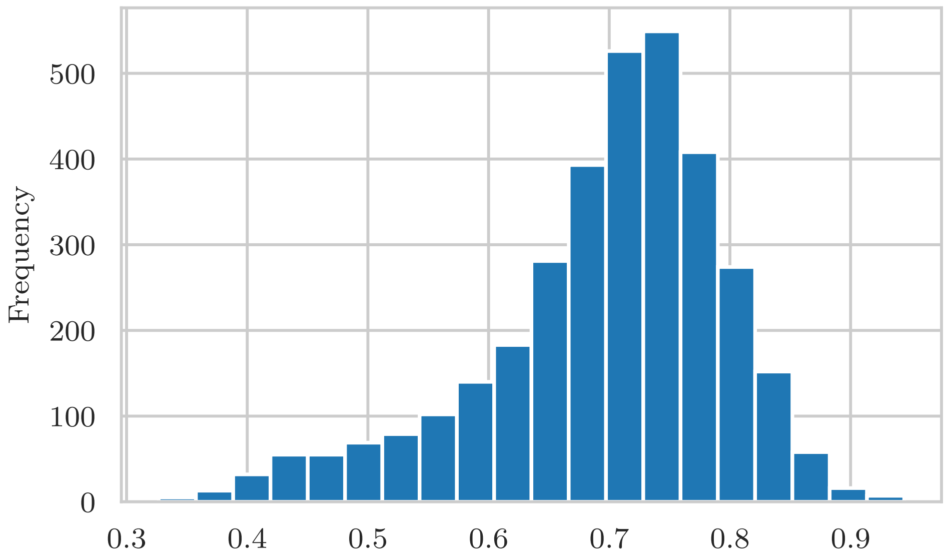


Figure 7: Distribution of Discount Factor $\hat{\beta}_i$

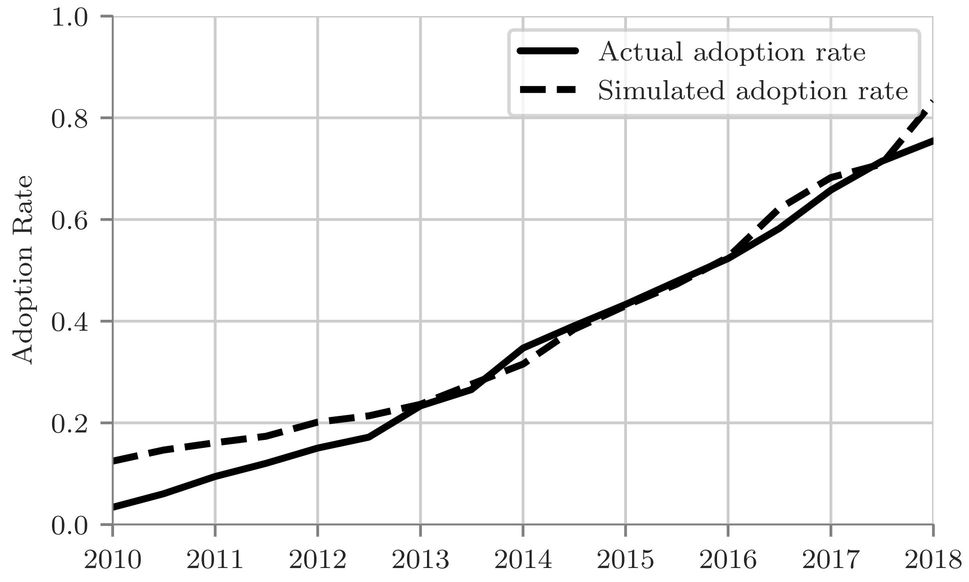


Figure 8: Actual vs. Simulated Adoption Rates

actual rate in Figure 8. The simulated adoption curve closely follows the actual one except in the first few periods, where the simulation over-predicts the adoption by 5% to 10%.

7 Counterfactuals

Katz and Shapiro (1985) predict that the success of a new technology and the speed of technology adoption highly depend on "sponsorship." A sponsor is an entity that is willing to make an investment to promote a new technology. There exists such a sponsor in the Python programming language—namely, Python Software Foundation (PSF). This is a nonprofit organization that oversees various issues in the Python community, including the transition from Python 2 to 3. Given a limited amount of resources, it is critical to understand how to efficiently allocate resources to avoid excess adoption inertia and promote a faster rate of adoption of Python 3.[34]

In this section, we examine the effectiveness of two counterfactual policies to promote faster Python 3 adoption: a lower fixed adoption cost and a community-level targeted subsidy. The first policy helps us to understand how much Python 3 adoption is affected by the fixed component of the adoption cost; the second one attempts to seek an "optimal" policy of cost subsidies to various sub-communities within Python.

7.1 Lower Fixed Adoption Cost

For this counterfactual exercise, we study how the Python 3 adoption decisions would respond to a lower fixed adoption cost c_0. As explained in Section 4.3, the fixed adoption cost

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

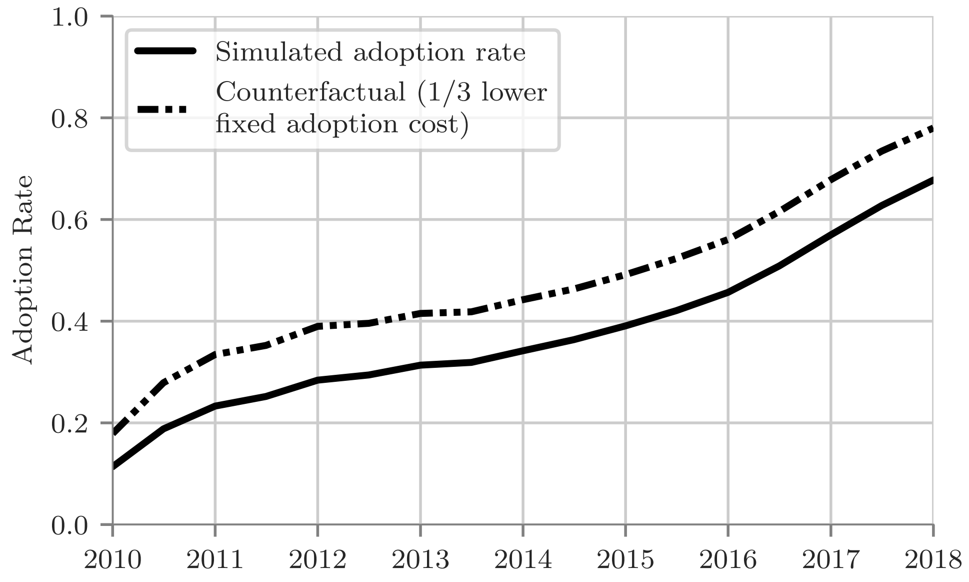


Figure 9: Simulated Adoption Rate with Lower Adoption Cost

comes from multiple sources, inlcuding differences in the language syntax and maintenance cost of future releases of a package. For PSF, there are multiple ways to achieve a lower c_0 for the entire Python community: for example, a better automatic conversion tool from Python 2 to 3 and a higher level of compatibility through scheduled deprecation.[35] We abstract away from the specific ways that PSF can lower c_0 and focus instead on the effect of a lower c_0 on the dynamics of Python 3 adoption over time.

Figure 9 contrasts the simulated adoption rates with and without the policy. Based on the state variables of 2010, we simulate adoption curves under two levels of fixed costs and draw the average adoption curve in Figure 9. We find that a one-third reduction in the fixed adoption cost leads to a roughly 12% increase in adoption rate by the end of 2018. The gap is small at the beginning, and it grows and stabilizes by 2018 as it gets closer to 100%. In particular, to reach an adoption rate of 40%, the cost reduction can help accelerate by two years from 2015 to 2013.

7.2 Community-Level Targeted Subsidy

The first counterfactual exercise studies the effects of a lower adoption cost for the whole Python community. In reality, lowering adoption cost for the whole Python programming language might be difficult due to a high development cost. An alternative way to promote Python 3 is through cost subsidies. Python has been used by many domains, and each forms its own community through specialized packages. Given limited resources, PSF can focus on providing subsidies to certain communities that are most reluctant to adopt Python 3 without

[35]We do not study the optimal compatibility decision made by PSF in this paper. The official reason for the incompatibility between Python 2 and 3 is the high development cost. In this paper, we focus on measuring the benefits of a lower adoption cost.

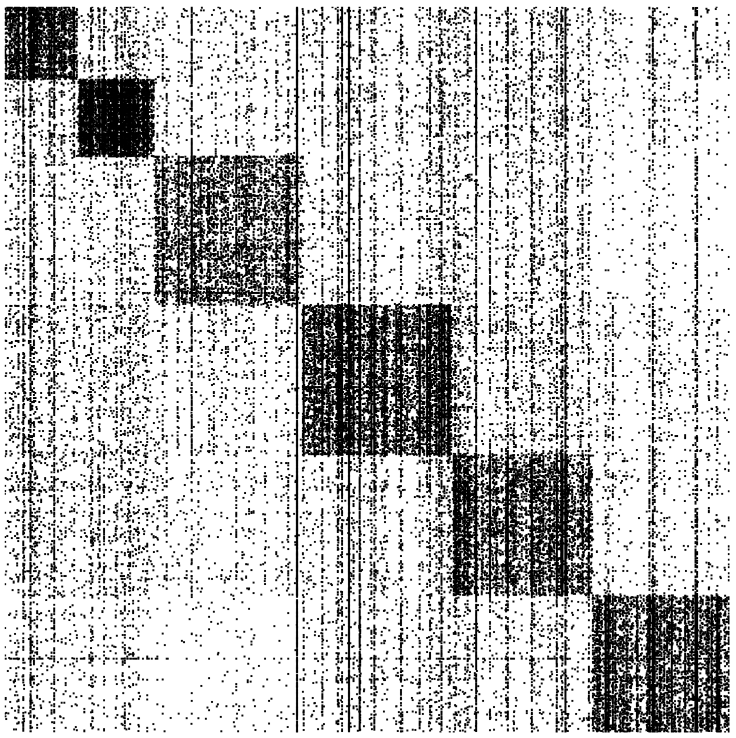


Figure 10: Python Community through Dependency Network

such a subsidy. In this subsection, we will investigate how subsidies to one community can affect its adoption rate over time, as well as its propagation effect on other connected communities.[36]

7.2.1 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 domain can customize it for their own use (for example, 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 their specific fields. Those packages that serve users within a given community tend to be more closely linked through dependencies.

The Louvain community detection algorithm (Blondel et al. (2008)) helps us to re-adjust the adjacency matrix, which clusters packages into six major communities. Figure 10 plots the updated adjacency matrix. Each dot represents a dependency between two packages, and each square represents a community: a larger one means more packages and a darker one shows denser links among packages within that community. Packages are more densely linked within the community and less so across communities. A handful of packages (those with

[36]Another example is a similar subsidy that can be found in artificial intelligence (AI). AI adoption can be beneficial in many fields, but how should the government allocate its promotion resources? See https://esrc.ukri.org/funding/funding-opportunities/canada-uk-artificial-intelligence-initiative-building-competitive-and-resilient-economies-and-societies-through-responsible-ai/ for a recent example.

long vertical lines) are used as dependencies by many packages across different communities.

Table 6: Package Characteristics by Community

ID	Description	Num of Packages	Avg Downloads	Avg Size (in KB)	Avg Num of Dependencies	Avg Num of Downstream Pkg
1	database	374	35325	301.851	2.891	2.431
2	cryptography	395	64846	362.081	3.621	2.289
3	web development	684	28189	224.363	2.573	2.215
4	web navigation	405	55461	203.544	2.721	1.982
5	software development	464	32870	368.818	3.194	1.549
6	data analysis	370	16782	777.119	3.091	2.420

Table 6 describes the functionalities and characteristics of packages for each community. In the Appendix, Figure 11 plots the network structure for each of the communities.

7.2.2 Policy Evaluation

We examine the effectiveness of community-level targeted subsidies on the adoption rates through both direct effects within that community and indirect effects on other communities, because communities are all linked in different ways. For each community, we adopt a cost subsidy policy that lowers the fixed adoption cost by one-third starting from 2010. We run the simulation of such policies for each of the seven communities 100 times each. Then we compare the differences between community-level adoption rates with and without the subsidy.

Table 7: Effectiveness of Community-Level Subsidy (Year 3)

	1	2	3	4	5	6
1	**7.31%***	0.38%	0.19%	-0.12%	0.27%	-0.98%
2	0.42%	**9.23%***	-0.17%	-0.49%	-0.05%	0.14%
3	-0.54%	0.24%	**8.32%***	-0.21%	0.17%	-0.16%
4	**-0.87%**†	-0.50%	-0.12%	**8.10%***	0.10%	-0.08%
5	-0.77%	-0.59%	0.17%	-0.46%	**9.06%***	0.29%
6	-0.36%	0.55%	0.44%	-0.23%	0.09%	**10.03%***

Notes: significance level: * 5% † 10%
Each row represents the targeted community that receives a subsidy equal to one-third of the fixed adoption cost c_0. The column shows the changes of community-level adoption rates.
This table reports results three years after the cost subsidy.

The responses of community-level adoption rates to subsidies are reported in Table 7. Each row represents a targeted community that receives a subsidy equal to one-third of the fixed adoption cost c_0. Each column shows the changes of community-level adoption rates. For example, the subsidy received by community 1 has caused 7.31% higher adoption in community 1 and 0.38% higher in community 2 after three years. The diagonal elements represent the within-community effects and the off-diagonal elements are the across-community effects of the subsidy.

Based on the 100 simulations, we also report the significance rates. A significance of 5% means that more than 95 out of the 100 simulation results show that the policy accelerates adoption. The results show that the subsidy on a targeted community accelerates adoption in that community. However, the effects on other non-subsidized communities can be positive or negative. On the one hand, more packages in the targeted community adopting Python 3 can result in more adoption of their downstream packages in other communities due to fewer incompatible dependencies. On the other hand, the subsidy policy increases the adoption probability of packages in the targeted community; thus, downstream packages in other communities have more incentive to wait because it is more likely to have a lower adoption cost in the near future. Most of the significant results in across-community effects are actually negative, indicating that the latter effect dominates the former in the first three years.

One natural subsequent question is what is the "optimal" policy? Suppose PSF wants to maximize the overall Python 3 adoption, but it has only limited resources to subsidize costs. Which community should then receive it?

Table 8: Effectiveness of Community-Level Subsidy

Target Type	1	2	3	4	5	6
Year 1						
ΔAR_m	7.12%	7.98%	6.66%	7.21%	7.16%	9.35%
$\Delta OverallAR$	0.70%	0.46%	1.03%	0.77%	0.83%	0.44%
Year 2						
ΔAR_m	6.88%	9.21%	6.82%	8.73%	8.41%	10.79%
$\Delta OverallAR$	0.71%	0.58%	1.54%	0.81%	1.07%	0.77%
Year 3						
ΔAR_m	7.31%	9.23%	8.32%	8.10%	9.06%	10.03%
$\Delta OverallAR$	0.84%	0.73%	1.86%	0.82%	1.13%	0.63%
Year 5						
ΔAR_m	9.27%	9.88%	10.18%	9.20%	9.22%	8.59%
$\Delta OverallAR$	1.06%	1.11%	2.18%	1.13%	1.23%	0.84%
Year 8						
ΔAR_m	11.22%	11.35%	9.66%	9.48%	9.95%	10.58%
$\Delta OverallAR$	1.29%	1.37%	2.00%	1.28%	1.31%	1.59%

The solution to this question needs to account for both the direct effect of subsidy within that community and the indirect effect on other communities. Moreover, the temporal dimension also matters. Table 8 summarizes the effectiveness of community-level subsidy to the targeted community and the overall Python community over time. Each column represents the targeted community m; ΔAR_m represents the difference between the adoption rate in that year with and without the subsidy, and $\Delta OverallAR$ is the difference of overall adoption rates with and without a community-targeted subsidy. For example, subsidy to community 1 increases the community-level adoption by 7.12% and the overall rate by 0.70% after one year.

Certain communities have strong reactions to the subsidy, but adoption in other communities might stall due to the negative indirect effects. Targeting community 6 can raise its adoption rate by 9.35%, but it has a minimal impact of 0.44% for the overall adoption, whereas a subsidy to community 3 can achieve a much larger impact of 1.03% despite a smaller within-community effect of 6.66%. Similar results hold for the long term as well.

The "optimal" policy also depends on the temporal dimension. For example, to maximize the overall adoption, community 5 seems to be a better target for cost subsidy than community 6 in the short term (0.83% vs. 0.44% in the first year), but the result is reversed in the long term (1.31% vs. 1.59% in the eighth year).

8 Conclusion

Technological changes have been fundamental to 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.

Our research explores how technology adoption can be affected by network structures in a disaggregated input-output network. We build a structural model of technology adoption to capture the dynamic compatibility decisions of packages that are interlinked through a rich input-output network. Our dynamic model allows each agent to anticipate the future actions of others.

The estimation results show the importance of dynamics in our model setting. Compared with a static one, the dynamic model better captures the decision-making process generated by the interactions in the network. We find strong evidence that the adoption decisions of downstream players are significantly affected by their upstream counterparts. It not only gives a better fit, but also implies the interactive elements that can only be captured by a structural dynamic model.

Through counterfactual analysis, we investigate the optimal policy to accelerate adoption. Simulation results show that when providing subsidies to a certain community of the population, the optimal policy highly depends on the network of both the targeted community and the interaction with other communities. Moreover, the time horizon aspect is also an important dimension to evaluate the effectiveness of a policy. Our counterfactual results imply that policymakers should focus not only on the direct effect on the recipients of subsidies, but also on the indirect effect on the whole industry to determine the optimal policy.

Our research studies network structure and technology adoption in the setting of the Python programming language, a large OSS platform. Without explicit pecuniary rewards, it helps us to avoid other complications that come with prices and focus instead on the pure effects of the network. We believe that our structural framework can be easily applied to study other industries where prices can be assumed to be fixed.

Undoubtedly, our model has many limitations when applied to a more general network.

This model depends on the acyclical property of the dependency relationships, whereas many networks can be cyclical. We also do not model the strategic interactions among agents (Assumption 2), which can be of first-order importance in many settings. In that sense, our research raises more questions than it can answer. The answers to these more general questions require not only high-quality data for one or multiple industries but also the corresponding development of econometric modeling and estimation methods. We hope that, as one of the first papers to link dynamic models and input-output networks, our work can contribute to further development in this direction.

References

Abbring, Jaap H, and Øystein Daljord. 2019. “Identifying the Discount Factor in Dynamic Discrete Choice Models.” *Becker Friedman Institute for Research in Economics Working Paper.*

Acemoglu, Daron, Vasco M Carvalho, Asuman Ozdaglar, and Alireza Tahbaz Salehi. 2012. “The Network Origins of Aggregate Fluctuations.” *Econometrica*, 80(5): 1977–2016.

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.

Björkegren, Daniel. 2018. “The Adoption of Network Goods: Evidence from the Spread of Mobile Phones in Rwanda.” *The Review of Economic Studies*, 37: 1033–1060.

Blondel, Vincent D, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. “Fast Unfolding of Communities in Large Networks.” *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10): P10008–10012.

Cabral, Luis MB. 2011. “Dynamic Price Competition with Network Effects.” *The Review of Economic Studies*, 78(1): 83–111.

De Groote, Olivier, and Frank Verboven. 2019. “Subsidies and Time Discounting in New Technology Adoption: Evidence from Solar Photovoltaic Systems.” *American Economic Review*, 109(6): 2137–2172.

DeLong, J Bradford. 2002. “Productivity Growth in the 2000s.” *NBER Macroeconomics Annual*, 17: 113–158.

Farrell, Joseph, and Garth Saloner. 1985. “Standardization, Compatibility, and Innovation.” *The RAND Journal of Economics*, 16(1): 70–83.

Fershtman, Chaim, and Neil Gandal. 2008. “Microstructure of Collaboration: The Network of Open Source Software.” *SSRN Electronic Journal*, 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 Joanna Stavins. 2004. “Network Externalities and Technology Adoption: Lessons from Electronic Payments.” *The RAND Journal of Economics*, 35(2): 260–276.

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

Hall, Bronwyn, and Beethika Khan. 2003. “Adoption of New Technology.” *New Economy Handbook*, 1–21.

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

Hendel, Igal, 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.

Katz, Michael L, and Carl Shapiro. 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): 95–116.

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.* 563–616. University of Chicago Press.

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): 479–501.

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): 649–676.

Xu, Lei, Tingting Nian, and Luis MB Cabral. 2019. “What Makes Geeks Tick? A Study of Stack Overflow Careers.” *Management Science*, Forthcoming.

9 Appendix

9.1 Examples of Python 3 New/Incompatible Features

Python 3 offers many major improvements and new features compared to Python 2. Some are compatible with Python 2 while others are not. The goal of this part of the Appendix is not to give a comprehensive comparison between Python 2 and 3, but rather to offer several examples to show the incompatibility between Python 2 and 3.

One of the most fundamental features that makes Python 3 backward incompatible is the default encoding system; namely, the way Python deals with text.

In Python 2, like all the classic programming languages such as C, Fortran, and Java, the encoding system is ASCII by default, which can basically only deal with English characters, punctuation, and digits that can be found on a standard English keyboard. Non-English characters have to be dealt with in a more complicated way. For example, typing "café" in Python 2 gives an error: "ascii" codec can't decode. There are several solutions to deal with this issue, but all require more advanced knowledge of the encoding system.

With the growing popularity of the Python programming language, especially among people who are new to programming, an easier way to deal with non-English characters has become one of the most requested features. Python 3 fundamentally changed its way of dealing with text by adopting Unicode as the default encoding system. With Python 3, "café" works perfectly well. However, much of the existing code in Python 2 fails to work, and many manual modifications need to be made.

Another fundamental change is the division of integers. In Python 2, like all other major programming languages, the division sign "/" means floor division; for example, $5/2 = 2$. Such behavior can be confusing for those new to programming. Python 3 changes it to behave more "naturally": $5/2 = 2.5$. The problem is that the old code in Python 2 works in Python 3 without errors but returns a different result.

9.2 Python Packages

As mentioned in Section 2, a package, in simple terms, can be defined as a collection of functions that anyone can use to finish more complex tasks. In this section, we provide a minimal example of Python code to illustrate what one can do with a package on Python.

A. Python Only

```
A = [1,2,3]
B = [1,1,0]
result = 0
for i in range(3):
    result = result + A[i] * B[i]
```

B. With Package NumPy

```
import numpy
A = numpy.array([[1,2,3]])
B = numpy.array([1,1,0])
A @ B
```

One can multiply two matrices $A = \begin{pmatrix}1\\2\\3\end{pmatrix}$ and $B = \begin{pmatrix}1\,1\,0\end{pmatrix}$ with dimensions 3x1 and 1x3 respectively using Python by creating two lists, looping through the two lists, and then summing up the multiplied value; that is, $1 \times 1 + 2 \times 1 + 3 \times 0 = 3$. Alternatively, one can use the NumPy package to first define two matrices and then multiply two matricies directly with $A@B$. The latter method is more elegant, less prone to error, and more computationally efficient, especially with large and multi-dimensional matrices.

9.3 Empirical Evidence for Assumption 1

The main reason for making the sequential-move assumption is that dependency packages often pre-announce their plans for future releases. The simplest way to verify this is to calculate the frequency of pre-announcement by upstream/dependency vs. downstream packages. Such information is often easy to find under the "roadmap" section on each package's website. However, this piece of information is not systematically recorded in a central depository. At the same time, for a limited number of packages, the roadmap information is provided under the Description section on PyPI. Therefore, instead of scraping all the packages' websites, we simply count the occurrences of "roadmap" found in the Description section for all the upstream-downstream package pairs. We don't think this crude measure would differ much from a more accurate data collection from scraping all the websites.

Table 9: Number of Occurrences with Roadmap Information

Both have roadmap	17
Only upstream/dependency package has roadmap	830
Only downstream package has roadmap	75
Neither has roadmap	5,285

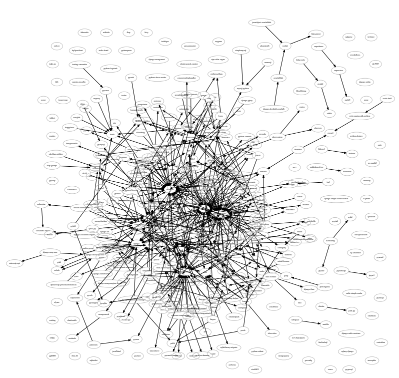
Table 9 lists the number of occurrences for each of the four scenarios. In most cases (5,285 occurrences), neither the upstream nor the downstream packages has roadmap information, probably due to the crudeness of this measure (i.e., roadmap information is provided on their own website instead of on PyPI). Conditional on having some roadmap information, it is mostly likely the case that only the dependency package has the information (830 occurrences), which supports the sequential-move assumption based on the dependency relationships.

9.4 Network Structure of Python Communities

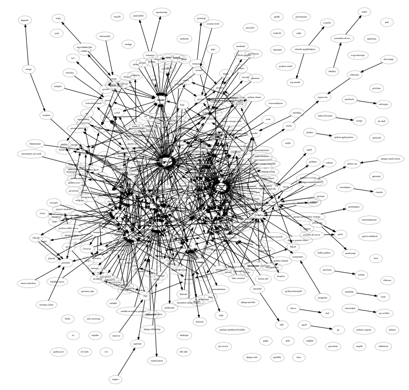
9.5 More Results on Counterfactual

Table 7 reports the effectiveness of community-level cost subsidy three years after the policy. In this section, instead of listing all the results across different years, we combine all the data and seek some systematic pattern in the effects of the counterfactual policy. Moreover, we measure the extent to which the prediction from a nonlinear model can be captured by a linear one. We do so by running the following regressions:

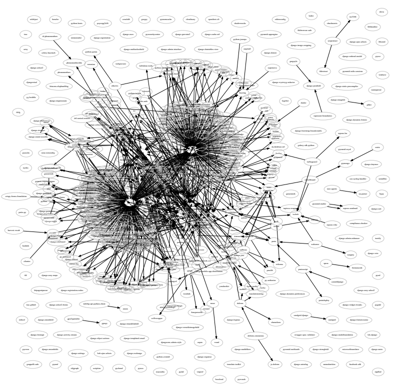
$$\begin{aligned}\Delta r_{mt} &= \gamma_0 + \gamma_1 Year_t + \gamma_2 Year_t^2 + \gamma_3 Density_m + \gamma_4 NumPackage_m + \gamma_5 AvgNumUpstream_m \\ &\quad + \gamma_6 AvgNumDownstream_m + \gamma_7 AvgDownload_m + \gamma_8 AvgSize_m + \epsilon_{mt} \\ \Delta r_{m'mt} &= \gamma_0 + \gamma_1 Year_t + \gamma_2 Year_t^2 + \gamma_3 Density_{m'm} + \gamma_4 NumPackage_m + \gamma_5 AvgNumUpstream_m \\ &\quad + \gamma_6 AvgNumDownstream_m + \gamma_7 AvgDownload_m + \gamma_8 AvgSize_m + (FE_{m'}) + \epsilon_{mt},\end{aligned}$$



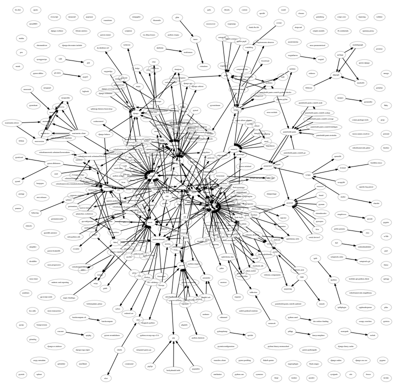
(a) 0-database



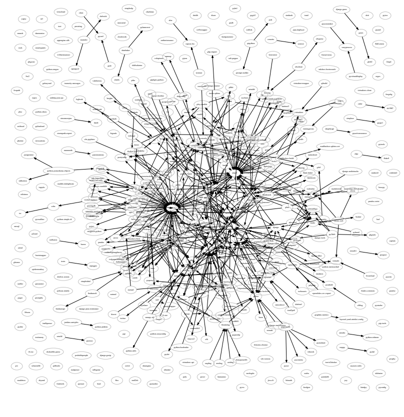
(b) 1-cryptography



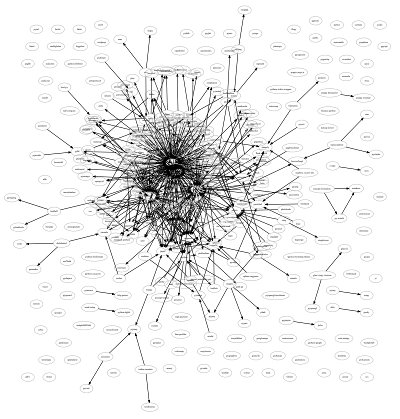
(c) 2-web development



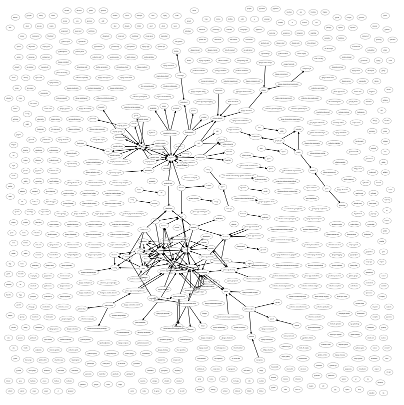
(d) 3-web navigation



(e) 4-software development & text processing



(f) 7-data analysis



(g) 99-other

Figure 11: Network Structure of Python Communities

where the dependent variables are defined as $\Delta r_{mt} = \widetilde{r}_{mt}$(subsidy on m)$-\widetilde{r}_{mt}$(without subsidy) and $\Delta r_{m'mt} = \widetilde{r}_{mt}$(subsidy on m') $- \widetilde{r}_{mt}$(without subsidy), respectively. Δr_{mt} measures the direct effect of subsidy on that community, and $\Delta r_{m'mt}$ measures the indirect effect of subsidy to m' on other communities m. Both are calculated as the differences between simulated adoption rate with and without the subsidy. The values of Δr_{mt} and $\Delta r_{m'mt}$ where t equals to year 3 correspond to the diagonal and off-diagonal values of Table 7. On the right-hand side of the equation, we control for some basic characteristics of a community. $Density_m$ measures the density of links within a community, and this is calculated as the number of actual links divided by the total number of potential links (all the possible combinations); $Density_{m'm}$ measures the density of links between two different communities, and it is calculated as the number of links originating from packages in community m' to those in community m, divided by the number of potential links. $NumPackage_m$ represents the total number of packages in the affected community; $AvgDownload_m$ and $AvgSize_m$ are the log value of average downloads and size in the affected community.

Table 10: Evaluation of Community-Level Subsidy by Linear Model

	A: Within-Community		B: Across-Community		
	(1)	(2)	(3)	(4)	(5)
	Δr_{mt}	Δr_{mt}	$\Delta r_{m'mt}$	$\Delta r_{m'mt}$	$\Delta r_{m'mt}$
Density$_m$	0.006	-0.025			
	(0.05)	(0.07)			
Density$_{m'm}$			0.094***	0.062*	-0.052
			(0.03)	(0.03)	(0.06)
NumPackage$_m$		0.000		0.000***	0.000***
		(0.00)		(0.00)	(0.00)
AvgDownload$_m$		0.003		0.004***	0.004***
		(0.00)		(0.00)	(0.00)
AvgSize$_m$		0.011**		0.004***	0.004***
		(0.01)		(0.00)	(0.00)
Year$_t$	0.012***	0.012***	-0.000	-0.000	-0.000*
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Year$_t^2$	-0.001***	-0.001***	0.000**	0.000***	0.000***
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Constant	0.057***	-0.042	-0.001***	-0.069***	-0.072***
	(0.00)	(0.07)	(0.00)	(0.01)	(0.01)
FE$_m$					x
N	119	119	714	714	714
R^2	0.699	0.725	0.032	0.141	0.165

The regression results are reported in Table 10. Note that the goal here is not to establish causal inference but rather to detect systematic patterns between community characteristics and effects from counterfactual policies, as well as to test how well a linear model captures variations generated from a structural model. In Panel A, the network density measure is not statistically significant. However, the variations of the effects for different communities are well captured by the whole linear model (R^2 around 70%). In Panel B, columns 3 and 4 indicate a positive correlation between network density and the effect on a community; that is, if two communities are densely connected, then a subsidy to one community is also more likely to have a larger effect on the other one. However, the estimates, after controlling for fixed effects of the targeted community m', seem to suggest that such effects do not exist, and that the aforementioned effects are likely to come from differences across the targeted communities instead of differences in the across-community densities. Note that the R^2 is rather small (16.5% even with the community-level fixed effects). These results indicate that though a linear model can well capture the variations of the direct effect, it fails to capture the indirect effect—namely, the effect that a policy has on other connected communities. These indirect effects can be large, as discussed in Section 7, and should be considered by policymakers in order to maximize the effectiveness of a promotion policy.