A Strategic Model of Software Dependency Networks

PRELIMINARY - COMMENTS WELCOME *

Co-Pierre Georg[1] and Angelo Mele[2]

[1]EDHEC Business School

[2]Johns Hopkins University - Carey Business School

Modern software development involves collaborative efforts and re-use of existing software packages and libraries, to reduce the cost of developing new software. However, package dependencies expose developers to the risk of contagion from bugs or other vulnerabilities. We study the formation of dependency networks among software packages and libraries, guided by a structural model of network formation with observable and unobservable heterogeneity. We estimate costs, benefits and link externalities of a package maintainer, using a scalable algorithm and data from 1,131,342 dependencies of 17,081 packages of the `Rust` programming language. We find evidence of a positive externality created by coders on other coders through the creation of dependencies. We also find that homophily and competition motives coexist in the creation of the network.

Keywords: software development, dependency graphs, strategic network formation, exponential random graphs

JEL Classification: D85, L17, L86, O31

* E-Mail: co-pierre.georg@edhec.edu and angelo.mele@jhu.edu.

1 Introduction

Modern software development is a collaborative effort that results in sophisticated software packages that makes extensive use of already existing software packages and libraries. This results in a complex network of dependencies among software packages, best described as dependency graphs.[1] While this results in significant efficiency gains for software developers (Lewis et al., 1991), it also increases the risk that an error in one software package, a bug, affects a large number of other software packages. A recent estimate by Krasner (2018) put the cost of losses from software failures at over USD 1 Trillion, up from USD 59 Billion in 2002, estimated by National Institute of Standards and Technology (2002). The number of reported incidents in the Common Vulnerabilities and Exposure Database has increased from 3,591 for the three years from 1999 to 2001 to 43,444 for the three-year period from 2017 to 2019—an almost twelve-fold increase.[2]

Because software bugs are not contained and can spread to all packages that depend on the faulty package, a software developer, which we also call a “coder", can exert an externality on other coders when reusing existing code. Whether this is the case and how large this externality is, is ultimately an empirical question. We model the decisions of developers to form a dependency to another package as an equilibrium game of network formation with observed and unobserved heterogeneity. The equilibrium characterization provides a likelihood of observing a particular dependency network architecture in the long run. Using data on open source software projects in the `Rust` programming language, obtained from its software package manager `Cargo`, we obtain all $1,131,342$ dependencies among `Cargo`'s $17,081$ packages. We indeed find that a coder's decision to create a dependency exerts a positive externality on other coders: Highly interdependent software packages are likely to become even more interdependent. This means that it is particularly important to ensure that such packages are free of bugs that potentially can affect a large number of other software packages. Using a package's Activity, Maturity,

[1] Dependency graphs have been studied for many different programming languages using information from package managers. For an empirical comparison of the most common ones, see Decan et al. (2019).

[2] The Common Vulnerabilities and Exposure database is a public reference for known information-security vulnerabilities which feeds into the US National Vulnerability Database. See `https://cve.mitre.org/`.

Popularity, and Size as covariates, we find evidence consistent with homophily among software packages of different types and competition among software packages of the same type.

Luckily, open source software is an ideal laboratory to study dependency graphs. Since the actual source code of a software package can be inspected and modified, reuse is not only possible, it is encouraged. Modern software systems are not developed in isolation, but within an ecosystem of interdependent packages. A package manager ensures that code dependencies are automatically resolved so that when a user installs one package, all of the package's dependencies are also installed. There are many programming languages which provide software using package managers. We use all the software packages written in the programming language `Rust` and managed by the `Cargo` package manager.[3] Each software package has a version, consisting of a `MaJOR`, `MINOR`, and `PATCH` number. When a package is created, the version number of each dependency is added as a reference. This ensures that updating a dependency does not have unintended consequences. We use data from `https://libraries.io`, which includes a full list of all dependencies and their versions for each package version.

We model the development of software as a process where coders reuse existing code, creating a network of dependencies among software packages in the process. This is typical for today's prevalent object-oriented software development paradigm. Our model describes a system of N individual software packages. Each package is maintained by a single coder (maintainer) who decides which dependencies to form. Maintainers obtain a net benefit of linking to another package which depends on how active this package is maintained, how mature, popular, and large it is. We also allow a maintainer's utility to be affected by a local, i.e. type-specific, externality: since dependent packages have dependencies themselves, they are susceptible to bugs imported from other packages. So we assume that maintainers care about the direct dependencies of the packages they link to. This specification excludes externalities that are more than two links away as in DePaula et al. (2018), Mele (2017) and Mele and Zhu (forthcoming). The network of dependencies forms over time and in each period a randomly selected package needs an update, so the maintainer has to decide whether to form a link or update the software in-house. Before updating the link, it receives a random match quality shock. We

[3] `Cargo` is `Rust`'s official package manager. See here for the official documentation.

characterize the equilibrium distribution over networks as a mixture of exponential random graphs (Schweinberger and Handcock, 2015; Mele, 2022), which can decomposed into within- and between-types contribution to the likelihood.

Estimation of this model is complicated because the likelihood depends on a normalizing constant that is infeasible to compute in large networks; moreover, the model's unobserved heterogeneity has to be integrated out in the likelihood thus further complicating the computations. To alleviate these problems, we resort to an approximate two-steps approach. In the first step, we estimate the discrete types of the nodes through approximations of the likelihood via a variational mean-field algorithm for stochastic blockmodels (Vu et al., 2013; Dahbura et al., 2021). In the second step, we estimate the structural payoff parameters using a fast Maximum Pseudo-Likelihood Estimator (MPLE), conditioning on the estimated types (Babkin et al., 2020; Dahbura et al., 2021).[4]

Our main result is that coders exert a positive externality on other coders when creating a dependency on another package. Other coders are then also more likely to create a dependency with that package. As a result, packages that are already highly interdependent tend to become even more interdependent. This increases the risk that a bug or vulnerability in a single package has large adverse consequences for the entire ecosystem. One example of how a bug in one package impacted a significant part of critical web infrastructure is the infamous Heartbleed vulnerability, which affected a large number of systems (Durumeric et al., 2014). Heartbleed is a bug in the widely used SSL/TLS cryptography library, resulting from an improper input variable validation. When Heartbleed was disclosed, it rendered 25-55% of secure webservers vulnerable to data theft, including of the Canadian Revenue Agency, and Community Health Systems, a large US hospital Chain. Our model provides a further argument for ensuring the security of highly interdependent software packages. Not only can a bug in such a package affect a large fraction of the entire ecosystem, it is also likely that this fraction increases as time goes on.

[4]These methods are implemented in a highly scalable open source R package `lighthergm`, available on Github at: `https://github.com/sansan-inc/lighthergm`. See Dahbura et al. (2021) for details on the implementation.

Another important question in the study of network formation processes is whether there is homophily in some of the node's observables. We find evidence of *homophily* between different types of software packages in terms of their Maturity, Popularity, and Size. In other words, mature, large and popular software packages of one type are likely to depend on similar packages of another type. This is intuitive, because mature, popular, and large packages are likely to have a large user base with high expectations of the software and coders try to satisfy this demand by providing sophisticated functionality with the help of mature, popular, and large software packages of another type. One example of this is the inclusion of a payment gateway in an e-commerce application. The e-commerce application itself can be sophisticated and complex, like ebay is in the provision of their auction mechanism. For such an application it is particularly valuable to provide users with the ability to use a variety of different payment methods, including credit card, paypal, or buy now pay later solutions like Klarna. Our result is also in line with the recent trend in software development away from large monolithic applications and towards interconnected microservices.

Lastly, we identify what can be labelled a *competition* motif in coders' decision to link to software packages of the same type. We find that coders are less likely to link to packages of the same type of similar popularity and size. These packages are more likely to be direct competitors. On the other hand, coders are more likely to connect to similarly active and similarly mature software packages of the same type, which we interpret as evidence for strategic substitutes in the innovation process, in particular for relatively novel and active software packages.

Overall, our results help us better understand the various driving forces and motifs in software development and how they shape the dependency graph of the entire software ecosystem. Our paper relates to several strands of academic literature in both economics and computer science.

Our paper contributes to a growing literature on open source software, which has been an interest of economic research.[5] In an early contribution, Lerner and Tirole (2002) argue that coders spend time developing open source software—for which they are typically not compensated—as a signal of their skills for future employers. Likewise, one reason why companies contribute

[5]For an overview of the broad literature in the emerging field of digital economics, see Goldfarb and Tucker (2019).

to open source software is to be able to sell complementary services. Open source projects can be large and complex, as Zheng et al. (2008) point out. They study dependencies in the Gentoo Linux distribution, and show that the resulting network is sparse, has a large clustering coefficient, and a fat tail. They argue that existing models of network growth do not capture the Gentoo dependency graph well and propose a preferential attachment model as alternative.[6]

Because most Linux distributions are free and open source software, they were early examples of software ecosystems. The package manager model is nowadays adopted by most programming languages which makes it feasible to use dependency graphs to study a wide variety of settings. Kikas et al. (2017), for example, study the structure and evolution of the dependency graph of `JavaScript`, `Ruby`, and also `Rust`. The authors emphasize that dependency graphs of all three programming languages have become increasingly vulnerable to the removal of a single software package. An active literature studies the network structure of dependency graphs (Decan et al., 2019) to assess the vulnerability of a software ecosystem (see, for example, Zimmermann et al. (2019)).

These papers show the breadth of literature studying open source ecosystems and dependency graphs. However, the literature almost exclusively looks at stochastic networks, while we model the formation of dependencies as a coder's *strategic* choice in the presence of various and competing mechanisms that either increase or reduce utility.[7]

The theoretical literature on strategic network formation has pointed out the role of externalities in shaping the equilibrium networks (Jackson, 2008; Jackson and Wolinsky, 1996). Estimating strategic models of network formation is a challenging econometric task, because the presence of externalities implies strong correlations among links and multiple equilibria (Mele, 2017; Snijders, 2002; DePaula et al., 2018; Chandrasekhar, 2016; DePaula, 2017; Boucher and

[6] In earlier work, LaBelle and Wallingford (2004) study the dependency graph of the Debian Linux distribution and show that it shares features with small-world and scale-free networks. However, LaBelle and Wallingford (2004) do not strictly check how closely the dependency graph conforms with either network growth model.

[7] Blume et al. (2013) study how possibly contagious links affect network formation in a general setting. While we do not study the consequences of the externality we identify for contagion, this is a most worthwhile avenue for future research.

Mourifie, 2017; Graham, 2017, 2020). In this paper, we model network formation as a sequential process and focus on the long-run stationary equilibrium of the model (Mele, 2017, 2022). Because the sequential network formation works as an equilibrium selection mechanism, we are able to alleviate the problems arising from multiple equilibria.

Adding unobserved heterogeneity further complicates identification, estimation and inference (Graham, 2017; Mele, 2022; Schweinberger and Handcock, 2015). Other works have considered conditionally independent links without externalities (Graham, 2017, 2020; DePaula, 2017; Chandrasekhar, 2016), providing a framework for estimation and identification in random and fixed effects approaches. On the other hand, because link externalities are an important feature in this contexts, we move away from conditionally independent links, and model nodes' unobserved heterogeneity as discrete types, whose realization is independent of observable characteristics and network, in a random effect approach. We can thus adapt methods of community discovery for random graphs to estimate the types, following approximations of the likelihood suggested in Babkin et al. (2020) and improved in Dahbura et al. (2021) to accomodate for observed covariates. Our two-steps method scales well to large networks, thus improving the computational challenges arising in estimation of these complex models (Boucher and Mourifie, 2017; Vu et al., 2013; Bonhomme et al., 2019).

Our model is able to identify externalities as well as homophily (Currarini et al., 2010; Jackson, 2008; Chandrasekhar, 2016), the tendency of individuals to form links to similar individuals. On the other hand, our model can also detect heterophily (or competition) among maintainers. We also allow the homophily to vary by unobservables, while in most models the homophily is estimated only for observable characteristics (Currarini et al., 2010; Schweinberger and Handcock, 2015; DePaula et al., 2018; Chandrasekhar, 2016; Graham, 2020).

2 A network description of code

2.1 Nomenclature and definitions

The goal of computer programs is to implement algorithms on a computer. An algorithm is a terminating sequence of operations which takes an input and computes an output using memory to record interim results.[8] We use the term broadly to include algorithms that rely heavily on user inputs and are highly interactive (e.g. websites, spreadsheet and text processing software, servers). To implement an algorithm, programming languages need to provide a means of reading input and writing output, have a list of instructions that can be executed by a computer, and provide a means of storing values in memory. More formally:

Definition 1. *Code is a sequence of operations and arguments that implement an algorithm. A computer program is code that can be executed by a computer system.*

In order to execute a program, a computer provides resources—processing power and memory—and resorts to a compiler or interpreter, which in themselves are computer programs.[9]

Software developers, which we refer to as coders, use programming languages to implement algorithms. Early programs were written in programming languages like `FORTRAN` and later in `C`, both of which adhere to the *procedural programming* paradigm. In this paradigm, code is developed via procedures that can communicate with each other via calls and returns.

Definition 2. *A procedure is a sequence of programming instructions, which make use of the resources provided by the computer system, to perform a specific task.*

[8] Memory to record interim instructions is sometimes called a “scratch pad”, in line with early definitions of algorithms which pre-date computers. See, for example, Knuth (1997) (Ch.1).

[9] The term "compiler" was coined by Hopper (1952) for her arithmetic language version 0 (A-0) system developed for the UNIVAC I computer. The A-0 system translated a program into machine code which are instructions that the computer can execute natively. Early compilers were usually written in machine code or assembly. Interpreters do not translate program code into machine code, but rather parse it and execute the instructions in the program code directly. Early interpreters were developed roughly at the same time as early compilers, but the first widespread interpreter was developed in 1958 by Steve Russell for the programming language LISP (see McCarthy (1996).

Procedures are an integral tool of almost all programming languages because they allow a logical separation of tasks and abstraction from low-level instructions, which greatly reduces the complexity of writing code. We use the term procedure broadly and explicitly allow procedures to return values. The first large software systems like the UNIX, Linux, and Windows operating systems or the Apache web server have been developed using procedural programming.

Today, however, the dominant modern software development paradigm is *Object-oriented programming* (OOP).[10] Under this paradigm, code is developed primarily in *classes*. Classes contain data in the form of variables and data structures as well as code in the form of procedures (which are often called methods in the OOP paradigm). Classes can communicate with one another via procedures using calls and returns.

Definition 3. *A class is a code template defining variables, procedures, and data structures. An object is an instance of a class that exists in the memory of a computer system.*

Access to a computer's memory is managed in most programming languages through the use of variables, which are memory locations that can hold a value. A variable has a scope which describes where in a program text a variable can be used.[11] The extent of a variable defines when a variable has a meaningful value during a program's run-time. Depending on the programming language, variables also have a type, which means that only certain kinds of values can be stored in a variable.[12]

Similar to variables, data structures are memory locations that hold a collection of data, stored in a way that implements logical relationships among the data. For example, an array is a data

[10] For a principal discussion of object-oriented programming and some differences to procedural programming, see Abelson et al. (1996). Kay (1993) provides an excellent historical account of the development of early object-oriented programming.

[11] There are three different types of variables in object-oriented code. First, a *member variable* can take a different value for each object of a class. Second, a *class variable* has the same value for all objects of a class. And third, a *global variable* has the same value for all objects (i.e. irrespective of the object's class) in the program.

[12] For example, `C` is a statically typed programming language, i.e. the `C` compiler checks during the compilation of the source code that variables are only passed values for storage that the variable type permits. Similarly, `Rust` is a statically typed programming language that, unlike `C`, is also object oriented. `Python`, on the other hand, is dynamically typed, and checks the validity of value assignments to variables only during run-time.

structure whose elements can be identified by an index. Different programming languages permit different operations on data structures, for example the appending to and deleting from an array.

Classes can interact in two ways. First, in the traditional *monolithic* software architecture, widely used for enterprise software like the one developed by SAP, for operating systems like Microsoft's Windows, and even in earlier versions of e-commerce platforms like Amazon, individual components cannot be executed independently. In contrast, many modern software projects use a *microservices* software architecture, which is a collection of cohesive, independent processes, interacting via messages. Both architectures result in software where individual pieces of code depend on other pieces, either within a single application or across various microservices. These dependencies form a network which we formalize in the next section.

2.2 Dependency Graphs

Actively maintained computer programs are frequently updated, which creates similar but different versions of the same program. To differentiate versions, coders use a semantic versioning notation, usually in the form `MAJOR.MINOR.PATCH`. Here, `MAJOR` indicates that a change led to incompatible API changes, while a change in the `MINOR` number denotes the addition of backward-compatible changes. An increase in the `PATCH` number indicates a bug fix. Kikas et al. (2017) discuss various ways of defining nodes and edges in a dependency graph and recommend using *actual* dependencies where each node is a specific version of the same program and links are actual dependencies between these versioned programs.[13] Note that nothing in this definition prevents classes from being bundled into libraries which provide additional logical and organizational structure.[14] To simplify the analysis, we assume that each library consists

[13] The other possibility is to treat each program as a node, subsuming all versions of the same program, and aggregate all dependencies between different versions of the same program. However, this method overstates the number of dependencies as Kikas et al. (2017) discuss.

[14] Libraries–also called modules or packages–are commonly developed by teams of coders and the literature on dependency graphs often looks at dependencies between libraries and then distinguish between internal and external dependencies of the library (see, for example, Decan et al. (2019)).

of exactly one program and each program consists of exactly one class developed by exactly one coder.[15]

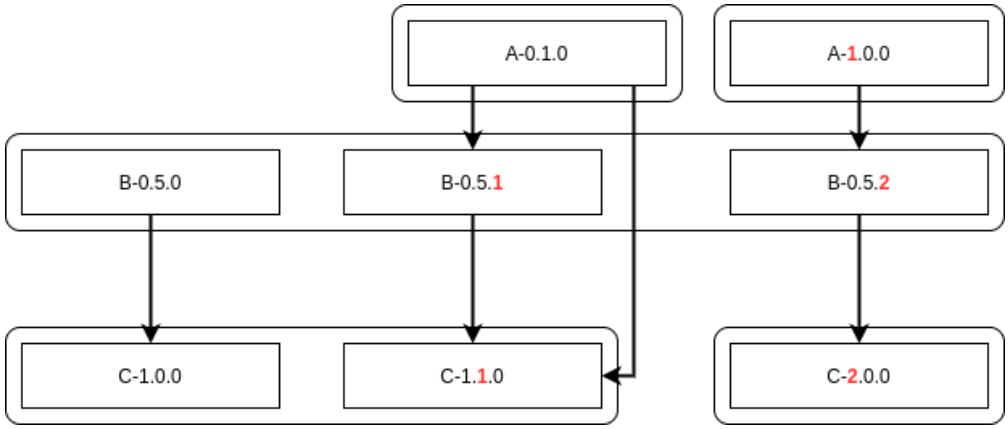
Consider a software system consisting of $\mathcal{N} = \{1, \ldots, n\}$ packages with $N = |\mathcal{N}| \geq 3$, each of which is developed by a different coder. Each package is identified by an identifier and a version number according to the above convention. A coder decides whether to implement all code for a package herself or to re-use existing code from other packages. This (re-)use of existing code gives rise to linkages between packages—one class depends on the other. The resulting network of packages and dependencies $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is called the *dependency graph* of the software system. Denote $g_{ij} = 1$ if $(i, j) \subseteq \mathcal{E}$ is an existing link from package i to package j, indicating that i depends on j. Denote as $g = \{g_{ij}\}$ the resulting $N \times N$ adjacency matrix of the dependency graph $\mathcal{G}$.

A consequence of the above definitions is that dependency graphs are specific to a given programming language. Such *software ecosystems* have grown tremendously over time. Decan et al. (2019), for example, show that the number of packages managed by the `npm` package manager for JavaScript has grown exponentially between 2012 and 2017. In 2020, `npm` contained over 1.3 million packages totalling 75 billion monthly downloads.[16] An example of a dependency graph for three programs "A", "B", and "C" with different versions following the semantic versioning notation is shown in Figure 1. Programs that share the same API can be included in exactly the same way from dependent programs. The figure includes an inherent time direction from left to right. Programs with lower `MAJOR`, `MINOR`, or `PATCH` version number are developed before those with higher version number. In Figure 1, B-0.5.0 depends on C-1.0.0, and A-0.1.0 depends on both B-0.5.1 and C-1.1.0, while B-0.5.1 depends on C-1.1.0. Then there is a major refactoring of both programs A and B, resulting in a dependency change where A-1.0.0 depends only on B-0.5.2 and B-0.5.2 depends on C-2.0.0 the new version of program C. The resulting dependency graph has eight nodes, six edges, and three connected components, of which (A-0.1.0, B-0.5.1, C-1.1.0) is the largest.

[15] An alternative approach is to study *call graphs* arising from procedures within the same software package (see, for example, Grove et al. (1997)).

[16] See *"GitHub nabs JavaScript packaging vendor npm"*. TechCrunch. Accessed 2021-02-03. (Source).

Figure 1: Dependency graph among three programs ("A", "B", "C") with different versions, following the semantic versioning notation. Programs with the same `MAJOR` number have the same API structure, indicated by a box with rounded edges. Arrows denote dependencies between the different versions of programs.



Realistic dependency graphs are large and growing in the number of packages and dependencies among them. JavaScript is the most popular programming language on GitHub, with over 2.7M repositories—each containing at least one package—hosted on the platform since 2017. To study the dependency graph of the Rust ecosystem, we use data from `https://libraries.io`, which includes all metadata extracted from the manifest of each Rust package included in the Cargo package manager.

There are three types of dependencies among projects in the data. Firstly, in an *exact* dependency requirement, a project with a given version depends on exactly one version of another project. This typically happens because not all projects are fully backward compatible. In these cases, it is safer for a coder to include a specific version of another project in her own code. About 18.2% of all dependencies are exact requirements. Secondly, *inequality* dependency requirements arise when a project with a given version depends on another project with a version larger than some minimal threshold. This usually is the case when a specific functionality has only been implemented in the dependency starting from a certain version, but all future versions of this project are backward compatible. It is then safe to use the latest version of the dependency, which can be beneficial if, for example, newer versions perform the same functionality with the same interface but faster. Inequality dependency requirements are used by

about 43% of all projects. Lastly, in a *wildcard* dependency requirement, a project depends on every version of another project. This usually happens if a specific functionality is so core to the dependency that it is included in all versions of the dependency. Wildcard dependencies are common, but also more prone to error propagation than exact dependencies. Roughly 30% of all dependency requirements are wildcard dependencies. These three types of requirements cover over 91% of all dependencies. The remaining requirements are combinations of inequality requirements and of inequality and wildcard dependency requirements.

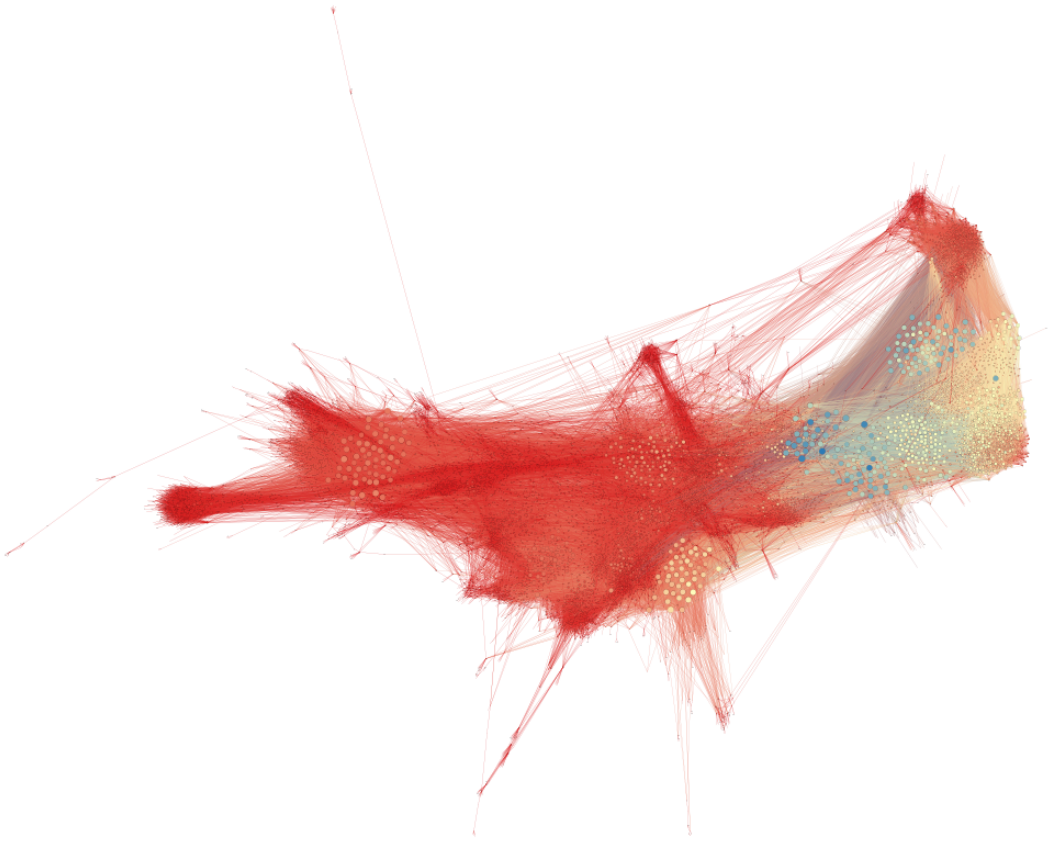
Figure 2 shows the dependency graph for the early sample. The size of each node indicates the number of projects depending on a project-version and the color indicates on how many projects a project-version depends (darker means more).

2.3 Covariate Data

In addition to the software dependency data, we also obtain additional data from the from `libraries.io` website for each package (e.g. `https://libraries.io/cargo/advapi32-sys`). First, we use the date of a package's first release and the total number of package releases to construct the average monthly number of releases, Activity, as a measure of how active the package is maintained. The time since the first release is a measure of a project's maturity in days, denoted Maturity. Next, to measure a package's popularity, we use the number of stars it has on github as an expression of how many people have liked the package there. We denote this measure Popularity. Lastly, the size of the package is measured in Byte and denoted Size. Since it's a direct measure for how much code has been written for a given project, larger projects require more effort by their developers.

To facilitate the estimation of our model, we create a categorical variable for each of our four covariates using quartiles of the distribution. The reason to discretize the variables is mostly computational, as our algorithm is based on stochastic blockmodels with discrete types (Bickel et al., 2013; Vu et al., 2013). If the covariates are discrete, then the whole machinery for estimation of blockmodels can be adapted to estimate the unobserved heterogeneity (Vu et al., 2013; Babkin et al., 2020; Dahbura et al., 2021); while with continuous variables the computational

Figure 2: Dependency graph for a 10% random sample of the Cargo Rust ecosystem. Each node is a project-version in the libraries.io dataset and each edge a dependency between two project-versions. Project-versions on which a greater number of other project-versions depend are larger. The dependency of project-versions on other projects ranges from red (few) to blue (many).



costs of estimation become prohibitive for such large networks.[17]

[17] We provide more details on this in the model and the appendix. More technical details are provided in Babkin et al. (2020) and Dahbura et al. (2021).

3 Model

We model the decisions of a software package's *maintainer*, a coder who is responsible to ensure that other coders' commits are reviewed and integrated properly and that the package is updated regularly. Our working assumption is that each package has exactly one maintainer and each maintainer is responsible for just one package. Since maintainers approve commits by individual coders, the decision to link to another package rests ultimately with the maintainer. This is a simplifying assumption, of course, since real packages can have more than one maintainer and many coders review and integrate code into a complex software package.

We consider an environment with $i = 1, ..., n$ packages and characterized by a vector of observable attributes $\boldsymbol{x}_i$, such as the Activity (monthly average number of commits), Maturity (time since inception of the package), Popularity (number of stars on github), and Size (lines of code). We also assume that there are K discrete types, unobservable to the researchers but observable to other maintainers, $\boldsymbol{z}_i = (z_{i1}, ..., z_{iK})$. The type of a software package can be thought of as the basic function of the package and examples include operating system components, graphical user interfaces, text processing software, compilers, and so on. A package of type k is denoted by $z_{ik} = 1$ and $z_{i\ell} = 0$ for all $\ell \neq k$. We use notation $\boldsymbol{x}$ and $\boldsymbol{z}$ to denote the matrix of observable and unobservable characteristics for all the packages.

The utility function of maintainer i from network $\boldsymbol{g}$, observables $\boldsymbol{x}$ and unobservables $\boldsymbol{z}$ is:

$$U_i\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}\right) = \sum_{j=1}^{n} g_{ij} u_{ij}(\alpha, \beta) + \sum_{j=1}^{n} \sum_{r \neq i,j}^{n} g_{ij} g_{jr} w_{ijr}(\gamma), \tag{1}$$

The payoff $u_{ij}(\boldsymbol{\alpha}, \boldsymbol{\beta}) := u\left(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{z}_i, \boldsymbol{z}_j; \boldsymbol{\alpha}, \boldsymbol{\beta}\right)$ is the direct utility of linking to package j. It is a function of observables $(\boldsymbol{x}_i, \boldsymbol{x}_j)$, unobservables $(\boldsymbol{z}_i, \boldsymbol{z}_j)$ and parameters $(\boldsymbol{\alpha}, \boldsymbol{\beta})$. This is the benefit of creating the dependencies to package j, net of costs—a maintainer will have to audit the code of the linked package and determine it's quality—of maintaining the link. The cost also includes modification and adaptation of the code that the maintainer has to do to be able to use the functions and methods available in package j. Furthermore, using another maintainer's code also increases the risk that the code contains an undetected bug (assuming that main-

tainers care more about the code of their own package than the code of their dependencies), which can be captured as a cost. And lastly, re-using code, while common practice in modern software development, means coders require fewer skills, which might negatively impact their future productivity and can thus be interpreted as another cost.

It is easy to see why package i including package j as a dependency creates a link from i to j, but there are also reasons why this creates a link from j to i: If package j is used in package i, it sends a signal to other maintainers considering which package to reference in their code. A package that is used by many other packages is likely to be of high quality.[18] Packages that are referenced in many other packages are also often seen as important to the community and thus receive more attention, including more proposals for bug fixes. Being referenced by many other packages also increases the visibility of a package, which can be important given the sheer amount of new packages that are created every month. On the other hand, there could also be a cost for package j if many other packages reference it. As increased attention leads to more commits and more proposed bug fixes, these proposals needs to be managed by a maintainer, which is costly. Being a highly referenced package might also increase expectations in terms of support and quality of documentation, which again carries a cost.

We assume that the utility function $u_{ij}(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is parameterized as follows

$$u_{ij}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \begin{cases} \alpha_w + \sum_{p=1}^{P} \beta_{wp} \mathbf{1}\{\boldsymbol{x}_{ip} = \boldsymbol{x}_{jp}\} & \text{if } z_i = z_j \\ \alpha_b + \sum_{p=1}^{P} \beta_{bp} \mathbf{1}\{\boldsymbol{x}_{ip} = \boldsymbol{x}_{jp}\} & \text{otherwise} \end{cases} \tag{2}$$

where the intercept α is interpreted as the cost of of forming the dependencies, and it is a function of unobservables only; and $\mathbf{1}\{\boldsymbol{x}_{ip} = \boldsymbol{x}_{jp}\}$ are indicators functions equal to one when the p-th covariates are the same for i and j. In this specification we allow the cost of a link to vary with unobserved heterogeneity, while the benefits vary with both observed and unobserved heterogeneity.

The second term of the utility function (1) concerns externalities from linking package j. In-

[18] The old adage in open source software that *"given enough eyes, all bugs are shallow"* goes even further and posits that a package that is used by sufficiently many other packages will have no bugs.

deed, package j may be linked to other packages r as well. This means that the maintainer needs to check the quality and features of the code in these packages, to make sure they are free of bugs and compatible with the maintainer's own code. On the other hand, any update to package r may compromise compatibility to package j and i, so this also includes a costs of maintaining. At the same time, if package j is linked by many packages, it means that it has an important functionality or it is of high quality. Whether this externality is positive or negative is ultimately an empirical question. We assume that this second term takes the form

$$w_{ijr}(\gamma) = \begin{cases} \gamma & \text{if } z_i = z_j = z_r \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

This specification assumes that the externality is local, that is only present if the link is within-types. In econometrics terms, this is a normalization that facilitates identification for parameter γ, through variation within types (Mele, 2022; Schweinberger and Handcock, 2015).

Summarizing, the maintainer needs to make decisions on whether to link another package or not. When making the decision the maintainer takes into account some externalities from this network of dependencies, but not all the possible externalities.

We assume a dynamic process of package development. Time is discrete and in each time period t only one maintainer is making decisions. At time t, a randomly chosen package i needs some code update. Package j is proposed for the update. If package j is not already linked to package i, we have $g_{ij,t-1} = 0$. If the maintainer decides to write the code in-house there is no update and $g_{ij,t} = 0$. On the other hand, if the maintainer decides to link to package j, the network is updated and $g_{ij,t} = 1$. If the dependency already exists (i.e. $g_{ij,t-1} = 1$), then the maintainer decision is whether to keep the dependency ($g_{ij,t} = 1$) or unlink package j and write the code in-house instead ($g_{ij,t} = 0$).

Formally, with probability $\rho_{ij} > 0$ package i coders propose an update that links package j. The maintainer decides whether to create this dependency or write the code in-house. Before linking the maintainer receives a random shock to payoffs $\varepsilon_{ij0}, \varepsilon_{ij1}$, which is iid among packages and time periods. This shock captures that unexpected things can happen both when develop-

ing code in house and when linking to an existing package.

So the maintainer i links package j if:

$$U_i\left(\boldsymbol{g}', \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}\right) + \varepsilon_{ij1} \geq U_i\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}\right) + \varepsilon_{ij0} \tag{4}$$

where $\boldsymbol{g}'$ is network $\boldsymbol{g}$ with the addition of link g_{ij}.

Throughout the paper we maintain the following assumptions (Mele, 2017):

1. The probability that coders of i propose an update that links to package j is strictly positive, $\rho_{ij} > 0$ for all pairs i, j. This guarantees that any dependencies can be considered.

2. The random shock to payoffs, ε_{ij}, follows a logistic distribution. This is a standard assumption in many random utility models for discrete choice.

As shown in Mele (2017) and Mele (2022), after conditioning on the unobservable types $\boldsymbol{z}$, the sequence of networks generated by this process is a Markov Chain and it converges to a unique stationary distribution that can be expressed in closed-form as a discrete exponential family with normalizing constant. In our model, the stationary distribution is

$$\pi(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}) \quad = \quad \prod_{k=1}^{K} \frac{e^{Q_{kk}}}{c_{kk}} \left[\prod_{l>k}^{K} \prod_{ij} \frac{e^{u_{ij}(\alpha_b, \beta_b)}}{1 + e^{u_{ij}(\alpha_b, \beta_b)}} \right], \tag{5}$$

where the potential function Q_{kk} can be written as:

$$\begin{aligned} Q_{kk} \quad &:= \quad Q\left(\boldsymbol{g}_{kk}, \boldsymbol{x}, \boldsymbol{z}; \alpha_w, \beta_w\right) \\ &= \quad \sum_{i=1}^{n} \sum_{j=1}^{n} g_{ij} z_{ik} z_{jk} \left(\alpha_w + \sum_{p=1}^{P} \beta_{wp} \mathbf{1}\{\boldsymbol{x}_{ip} = \boldsymbol{x}_{jp}\} \right) + \frac{\gamma}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{r \neq i,j}^{n} g_{ij} g_{jr} z_{ik} z_{jk} z_{rk}, \end{aligned} \tag{6}$$

and $\boldsymbol{g}_{kk}$ is the network among nodes of type k; the normalizing constant is:

$$c_{kk} = \sum_{\omega_{kk} \in \mathcal{G}_{kk}} e^{Q_{kk}}. \tag{7}$$

Here, $\mathcal{G}_{kk}$ is the set of all possible networks among nodes of type k.[19]

The stationary distribution π represents the long-run distribution of the network. The second part of (5) represents the likelihood of between-types links, while the first part is the likelihood of within-types links. This simple decomposition is possible because the externalities have been normalized to be local. So the model converges to K independent exponential random graphs within-types, and conditionally independent links between-types.

Because of this simple characterization of the stationary equilibrium, the incentives of the maintainers are summarized by a potential function, whose maxima are pairwise stable networks. Because of the externalities, in general the equilibrium networks will be inefficient, as they will not maximize the sum of utilities of the maintainers.[20]

3.1 Estimation

Estimation of this model is challenging because of the normalizing constants c_{kk} in the likelihood and the discrete mixture model for the block assignments. We bypass these issues by using recently developed approximate estimation methods that allow estimation of the block structure and the structural parameters in two steps.

In this section, we provide a general overview of the estimation procedure and refer the reader interested in the technical details to the Appendix and Dahbura et al. (2021).

Formally, we use a random effect approach where the unobservable types are independent of observables and the network. We assume that the types are drawn from the same multinomial distribution and are independent:

$$z_i \overset{iid}{\sim} Multinomial\left(1;\eta_1,\eta_2,\cdots,\eta_K\right) \tag{8}$$

[19] As noted in Mele (2017), the constant c_{kk} is a sum over all $2^{n_k(n_k-1)/2}$ possible network configurations for the nodes of type k. This makes the computation of the constant impractical or infeasible in large networks.

[20] This is evident when we compute the sum of utilities. Indeed, the welfare function will include all the externalities and therefore will be in general different from the welfare function.

Define $L\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}\right) := p_\eta(\boldsymbol{z})\pi(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma)$, where $p_\eta(\boldsymbol{z})$ is the multinomial in (8) and $\pi(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma)$ is the likelihoood in (5). Therefore, the log-likelihood of observing network $\boldsymbol{g}$ in the data is:

$$\begin{aligned}\mathscr{L}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}) &= \log \sum_{\boldsymbol{z}\in\mathscr{Z}} p_\eta(\boldsymbol{z})\pi(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma) \\ &= \log \sum_{\boldsymbol{z}\in\mathscr{Z}} L\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}\right) \qquad (9)\end{aligned}$$

To estimate the unobservable types, we consider an approximation of the likelihood based on the stochastic blockmodel (Babkin et al., 2020; Dahbura et al., 2021). In fact, the network formation model with $\gamma = 0$ corresponds to a stochastic blockmodel, and the log-likelihood (9) is well approximated by the likelihood of a stochastic blockmodel.[21] Therefore, we recover an approximate block structure using methods of community discovery for stochastic blockmodels.

In the first step we recover the unobserved types. We do this by approximating the likelihood using a stochastic blockmodel. In this step we do not estimate the structural parameters. We use a variational approximation to find a lower bound to the stochastic blockmodel likelihood. The rationale is that the lower bound is simpler to maximize and consistently estimates the types (Bickel et al., 2013; Wainwright and Jordan, 2008; Babkin et al., 2020; Mele and Zhu, forthcoming). Because of the large network size in our data, maximizing the variational lower bound is too slow from a computational point of view. We therefore follow the methods of Vu et al. (2013) and maximize a minorizer of the lower bound. A minorizer is a function that is easy to maximize, and it is always lower than the variational lower bound. Maximizing the minorizer at each iteration of the EM algorithm implies an improvement of the variational lower bound by construction. This maximization is highly parallelizable, involving n parallel maximization problems, thus improving computational efficiency. The variational EM algorithm iteratively updates the parameters and converges to a local maximum of the lower bound. We assign each node to the modal type obtained by the variational EM algorithm.

Once we obtain an estimate $\widehat{\boldsymbol{z}}$ of the types assigned to each node, we estimate the structural parameters of the utility functions using a Maximum Pseudo-Likelihood Estimator (MPLE) for

[21] The technical conditions under which this approximation works well are in Babkin et al. (2020).

exponential random graphs (Boucher and Mourifie, 2017; Snijders, 2002). This amounts to find the parameter vector that maximizes the product of conditional probabilities of links.

The identification of the within-types parameters is obtained by variation across types, while identification of the parameters between types follows the usual identification in logistic regression. Additional technical details relative to the consistency of types recovery are in Babkin et al. (2020); Schweinberger (2020); Schweinberger and Stewart (2020).

In summary the algorithm we use has the following two steps:

1. Run the variational EM algorithm with minorization approach to estimate the types of each node $\widehat{z}$;
2. Conditional on the estimated $\widehat{z}$, estimate the structural parameters $(\alpha_w, \alpha_b, \beta_w, \beta_b, \gamma)$ using a pseudolikelihood estimator.

All the derivations and the iterative procedure to find the lower-bound are in Appendix.[22]

3.2 Estimation Results

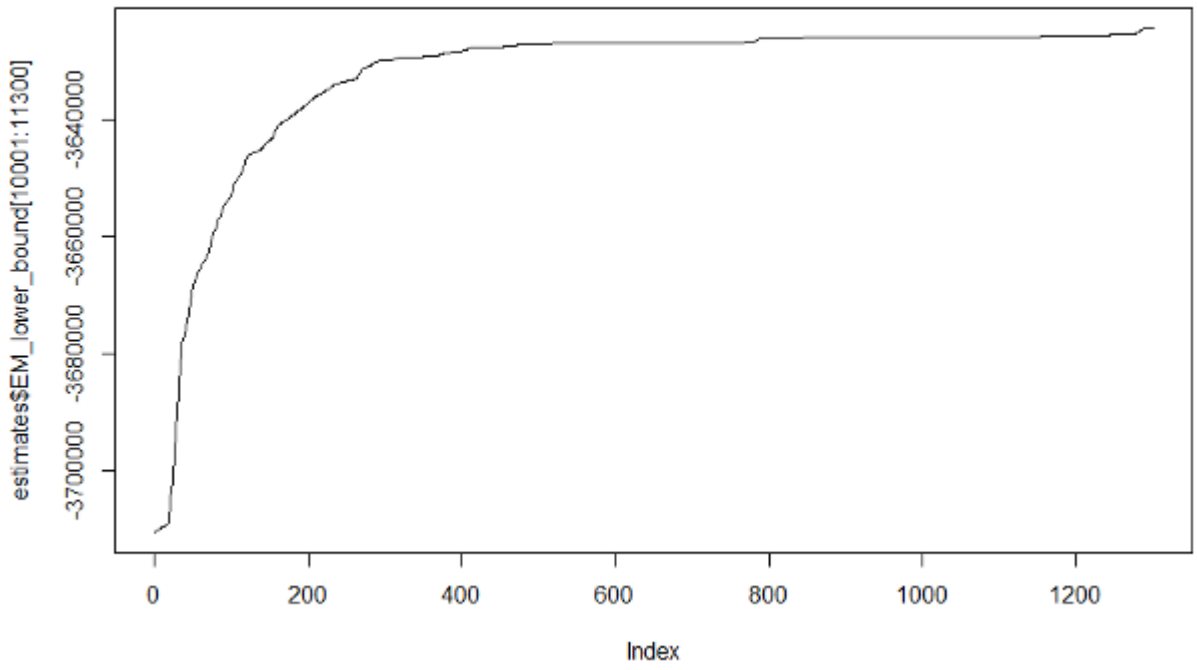
We estimate the model using $K = 50$ types.[23] We first run the variational EM algorithm for 10,000 steps, without using information on the covariates. This corresponds to a standard stochastic blockmodel with 50 blocks. We initialize the blocks using the InfoMap algorithm; this is very fast and provides a good starting value. We then run the variational EM algorithm for 10,000 steps, without using the information of the covariates, to speed up computations. Once the lower bound converges, we re-start it including the node covariates information. This last step uses more memory because of the the computation involving the matrix of covariates, and it is significantly slower.[24] We report the convergence of the variational lower bound in Figure 3, for the last 1300 steps, showing relatively good convergence. Once we achieve convergence for the lower bound, we obtain estimated types assignments $\hat{z}$, and estimate the structural parameters conditioning on $\hat{z}$. The estimated structural parameters are shown in Table 1.

[22] See also Dahbura et al. (2021) for explicit formulas on how to compute the probabilities in the Variational EM

Table 1: Estimated Parameters with $K = 50$ types. Estimates are obtained by Maximum Pseudo-Likelihood, conditioning on the estimated types in the first step of the estimation.

	Dependent variable:	
	BETWEEN	WITHIN
	(1)	(2)
Cost (α)	-5.920^{***}	-5.146^{***}
	(0.003)	(0.007)
Number of dependencies (γ)		0.018^{***}
		(0.00003)
Same Activity (β_1)	-0.117^{***}	0.110^{***}
	(0.002)	(0.007)
Same Maturity (β_2)	0.098^{***}	0.130^{***}
	(0.002)	(0.008)
Same Popularity (β_3)	0.675^{***}	-0.058^{***}
	(0.003)	(0.008)
Same Size (β_4)	0.728^{***}	-0.508^{***}
	(0.002)	(0.007)
AIC	11,095,280.000	800,350.800
BIC	11,095,364.000	800,433.600
Note:	*p<0.1; **p<0.05; ***p<0.01	

Figure 3: Convergence of the lower bound. The partition of nodes is initialized with InfoMap and 10000 iterations of the VEM algorithm without covariates, The figure show the likelihood lower bound in the last 1300 iterations of the VEM using covariate information.



First, since the terms of the utility function (1) are net benefits, the negative intercept for the estimation both between different types of software packages and among software packages of the same type is best interpreted as the maintainer's cost of developing a package. It depends on unobservables only, which is a reasonable assumption, since key determinants of the maintainer's cost, such as their ability, are unobservable to us.

Table 1 indicates that the cost of linking to a different-type package is higher than the cost of

algorithm.

[23] We will add robustness checks in future versions of the paper, as well as more formal tests for the choice of K.

[24] Each iteration of the variational EM takes ≈ 0.0001 seconds without covariates and ≈ 50 seconds with covariates. To save on memory usage we pre-compute a sparse matrix version of the covariates. This slows down computation as sparse matrix routines are usually slower than the corresponding dense matrix routines. However, the saving in terms of RAM is important for scalability to large networks.

linking to a same-type package. This is intuitive, since maintainers likely need to exert more effort to incorporate a package that has a very different functionality and that they are hence less familiar with. A good example of this kind of dependency is the use of low-level libraries for printing used by higher-level text processing software.

One of the key results of our estimation is the parameter γ, which indicates the size of the externality. Given our econometric setup, we obtain estimates for within-type linkages only, and these indicate that a maintainer's decision to link to a given package exerts a positive externality on other maintainers so that it is more likely that these will link to the package as well. In other words, maintainers prefer linking to packages of the same type that have more dependencies.

The positive and statistically highly significant value of γ implies that already highly interdependent packages tend to become even more interdependent. It is then intuitive that it is important to ensure that these packages are free of bugs because they otherwise could affect a disproportionately large number of other packages. And while a formal description of such a contagion process is beyond the scope of our paper, the implications for cybersecurity and the potential for business interruptions are clear.

Next, we turn to the covariates. Maintainers are more likely to link to similarly mature, popular, and complex packages of a different type. This is in line with the idea that more mature, popular, and complex packages are more likely to have a large user base and therefore have a greater obligation to provide a stable package. Achieving this is easier for links to mature, and popular, and complex–or sophisticated–packages.

At the same time, though, maintainers are less likely to link to similarly actively managed packages. This is perhaps because maintainers of more actively managed packages prefer to create dependencies to more stable software packages of a different type. One reason for this is that it is more costly for maintainers to understand software packages of a different type, a cost that is even higher if the linked package is under active development.

When considering packages of the same type, maintainers are more likely to link to similarly active and mature packages, but less likely to link to similarly popular and large packages.

We treat these results only as indicative, but they paint a consistent picture. Packages of the same type (and same maturity, popularity, and size) are more likely to be competing software packages (competing browsers, text processing software, web servers, etc.) and hence it is less likely that maintainers link to them, especially if the competitor package is similarly popular and complex.

To understand the positive coefficients β_1 and β_2 within the same type, it helps to better understand how we create the type classification. We have chosen the number of types exogenously and set it to $K = 50$. The clustering into types was done using our variational EM algorithm. But there is no guarantee that this exogenously chosen number is actually correct. So in the clustering to types, it is likely that there are large, mature, popular, and actively managed packages, but also those that are small, cutting edge, actively developed but not yet well known purpose-built packages. These are likely to depend on packages that are also actively maintained but not very mature.

In fact, the positive sign for the same maturity can also be understood when recognizing the temporal ordering of software updates. The nodes in our network were created at different points in time and it is very likely that relatively newer versions of a package (those with larger `MAJOR` number, for example) link to relatively new versions of another package. And on the other hand, it is physically impossible that, say a five year old version of a software package has a dependency that was released last month.

4 Conclusion

The network of dependencies among software packages is an interesting laboratory to study network formation and externalities. Indeed, the creation of modern software gains efficiency by re-using existing libraries; on the other hand, dependencies expose new software packages to bugs and vulnerabilities from other libraries. This feature of the complex network of dependencies motivates the interest in understanding the incentives and equilibrium mechanisms driving the formation of such networks.

In the present work, we developed a structural analysis of the motives, costs, benefits and externalities that a maintainer faces when developing a new software package. The empirical model allows us to disentangle observable from unobservable characteristics that affect the decisions to form dependencies to other libraries.

We find evidence that coders create positive externalities for other coders when creating a link. We also find interesting evidence that packages of the same type (and same maturity, popularity, and size) seem to compete among each other (competing browsers, text processing software, web servers, etc.), probably because they are substitutes in terms of the dependency graph.

This preliminary evidence raises more questions about the formation of dependency graphs among software developers. First, how does the estimated network formation model affect contagion and vulnerabilities of the system? The presence of an externality implies that maintainers may be inefficiently create dependencies, thus increasing the density of the network and the probability of contagion. The extent of this risk and the correlated damage to the system is of paramount importance and we plan to explore it in future versions. Second, while we have focused on a stationary realization of the network, there are important dynamic considerations in the creation of these dependencies. While the modeling of forward-looking maintainers may be useful to develop intuition about intertemporal strategic incentives and motives, we leave this development to future work. Finally, we have focused on a single language, but the analysis can be extended to other languages as well, such as `Java`, `C++`, etc.

References

Abelson, H., Sussman, G. J. and Sussman, J. (1996), *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA.

Babkin, S., Stewart, J., Long, X. and Schweinberger, M. (2020), 'Large-scale estimation of random graph models with local dependence'.

Bickel, P., Choi, D., Chang, X. and Zhang, H. (2013), 'Asymptotic normality of maximum likelihood and its variational approximation for stochastic blockmodels', *The Annals of Statistics* **41**(4), 1922 – 1943.

Blume, L., Easley, D., Kleinberg, J., Kleinberg, R. and Tardo's, E. (2013), 'Network formation in the presence of contagious risk'.

Bonhomme, S., Lamadon, T. and Manresa, E. (2019), 'A distributional framework for matched employer employee data', *Econometrica* **87**(3), 699–739.

Boucher, V. and Mourifie, I. (2017), 'My friend far far away: A random field approach to exponential random graph models', *Econometrics Journal* **20**(3), S14–S46.

Chandrasekhar, A. G. (2016), Econometrics of network formation, *in* yann bramoulle, andrea galeotti and brian rogers, eds, 'Oxford handbook on the economics of networks.', Oxford Univerisity Press.

Currarini, S., Jackson, M. O. and Pin, P. (2010), 'Identifying the roles of race-based choice and chance in high school friendship network formation', *the Proceedings of the National Academy of Sciences* **107**(11), 4857ï¿½4861.

Dahbura, J. N. M., Komatsu, S., Nishida, T. and Mele, A. (2021), 'A structural model of business cards exchange networks'.

Decan, A., Mens, T. and Grosjean, P. (2019), 'An empirical comparison of dependency network evolution in seven software packaging ecosystems', *Empirical Software Engineering* **24**, 381–416.

DePaula, A. (2017), Econometrics of network models, *in* B. Honore, A. Pakes, M. Piazzesi and L. Samuelson, eds, 'Advances in Economics and Econometrics: Eleventh World Congress', Cambridge University Press.

DePaula, A., Richards-Shubik, S. and Tamer, E. (2018), 'Identifying preferences in networks with bounded degree', *Econometrica* **86**(1), 263–288.

Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M. and Halderman, J. A. (2014), The matter of heartbleed, *in* 'Proceedings of the 2014 Conference on Internet Measurement Conference', IMC '14, Association for Computing Machinery, New York, NY, USA, pp. 475–488.

Goldfarb, A. and Tucker, C. (2019), 'Digital economics', *Journal of Economic Literature* **57**(1), 3–43.

Graham, B. (2017), 'An empirical model of network formation: with degree heterogeneity', *Econometrica* **85**(4), 1033–1063.

Graham, B. (2020), Network data, *in* S. Durlauf, L. Hansen, J. Heckman and R. Matzkin, eds, 'Handbook of econometrics 7A', Amsterdam: North-Holland,.

Grove, D., DeFouw, G., Dean, J. and Chambers, C. (1997), Call graph construction in object-oriented languages, *in* 'Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications', OOPSLA '97, Association for Computing Machinery, New York, NY, USA, p. 108–124.

Hopper, G. M. (1952), The education of a computer, *in* 'Proceedings of the Association for Computing Machinery Conference', pp. 243–249.

Jackson, M., ed. (2008), *Social and economic networks*, Princeton.

Jackson, M. O. and Wolinsky, A. (1996), 'A strategic model of social and economic networks', *Journal of Economic Theory* **71**, 44–74.

Kay, A. C. (1993), The early history of smalltalk, Url: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html.

Kikas, R., Gousios, G., Dumas, M. and Pfahl, D. (2017), Structure and evolution of package dependency networks, *in* 'Proceedings - 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR 2017', pp. 102–112.

Knuth, D. E. (1997), *The art of computer programming, volume 1 (3rd ed.): Fundamental algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Krasner, H. (2018), The cost of poor quality software in the us: A 2018 report, Report.
URL: *https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-repo The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf*

LaBelle, N. and Wallingford, E. (2004), 'Inter-package dependency networks in open-source software'.

Lerner, J. and Tirole, J. (2002), 'Some simple economics of open source', *The Journal of Industrial Economics* **50**(2), 197–234.

Lewis, J. A., Henry, S. M., Kafura, D. G. and Schulman, R. S. (1991), An empirical study of the object-oriented paradigm and software reuse, *in* 'OOSPLA 91', pp. 184–196.

McCarthy, J. (1996), The implementation of LISP, Stanford AI Lab: The History of LISP.
URL: *http://www-formal.stanford.edu/jmc/history/lisp/lisp.html*

Mele, A. (2017), 'A structural model of dense network formation', *Econometrica* **85**(3), 825–850.

Mele, A. (2022), 'A structural model of homophily and clustering in social networks', *Journal of Business & Economic Statistics* **40**(3), 1377–1389.

Mele, A. and Zhu, L. (forthcoming), 'Approximate variational estimation for a model of network formation', *Review of Economics and Statistics* .

National Institute of Standards and Technology (2002), Software errors cost u.s. economy $59.5 billion annually, Press release.
URL: *https://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm*

Schweinberger, M. (2020), 'Consistent structure estimation of exponential family random graph models with block structure', *Bernoulli* **26**, 1205–1233.

Schweinberger, M. and Handcock, M. S. (2015), 'Local dependence in random graph models: characterization, properties and statistical inference', *Journal of the Royal Statistical Society–Statistical Methodology Series B* **77**, 647–676.

Schweinberger, M. and Stewart, J. (2020), 'Concentration and consistency results for canonical and curved exponential-family models of random graphs', *Annals of Statistics* **48**(1), 374–396.

Snijders, T. A. (2002), 'Markov chain monte carlo estimation of exponential random graph models', *Journal of Social Structure* **3**(2).

Vu, D. Q., Hunter, D. R. and Schweinberger, M. (2013), 'Model-based clustering of large networks', *The Annals of Applied Statistics* **7**(2), 1010 – 1039.

Wainwright, M. and Jordan, M. (2008), 'Graphical models, exponential families, and variational inference', *Foundations and Trends@ in Machine Learning* **1**(1-2), 1–305.

Zheng, X., Zeng, D., Li, H. and Wang, F. (2008), 'Analyzing open-source software systems as complex networks', *Physica A: Statistical Mechanics and its Applications* **387**(24), 6190–6200.

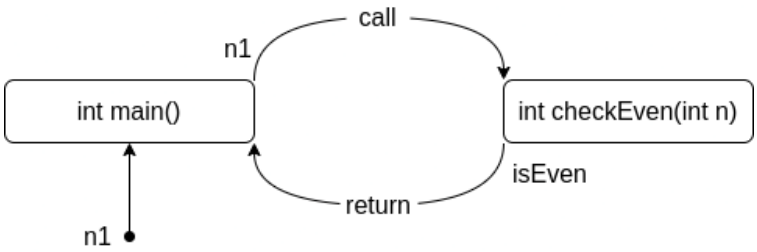
Zimmermann, M., Staicu, C.-A., Tenny, C. and Pradel, M. (2019), Small world with high risks: A study of security threats in the npm ecosystem, *in* '2019 USENIX Security Symposium, USENIX Security '19'.

A Example Code

A.1 A simple example of procedural programming

The most widely used procedural programming language is `C`, developed in the early 1970s by Dennis Ritchie. The `C` compiler takes the program source code and translates it into machine-executable code, specific to a given computer system. A simple example of procedural programming code to read in an integer and check if it is positive can be found in Figure A2. The program starts when the `main()` function is executed. It then reads in an integer `n1` from the terminal prompt and checks whether this number is even. The check is performed as a function call to the `int checkEven(int n)` function which takes an integer `n` and returns itself an integer.

Figure A1: The "CheckEven" program depicted as a network.



Notes: Each function is a node and a call/return pair links the two functions. The `main()` function reads a variable `n1` from the prompt and passes it to the `checkEven(int n)` function, which returns an integer `isEven` which equals one if the variable `n` is even. Variables that are passed in a call/return pair are shown at the beginning of the connecting arrow. The program code is detailed in Figure A2.

A.2 A simple example of object-oriented programming

Similar to procedural programming, it is possible to map objects and their relations defined in a piece of code into the nodes and links of a network representing this code. This is captured in Definition 3.

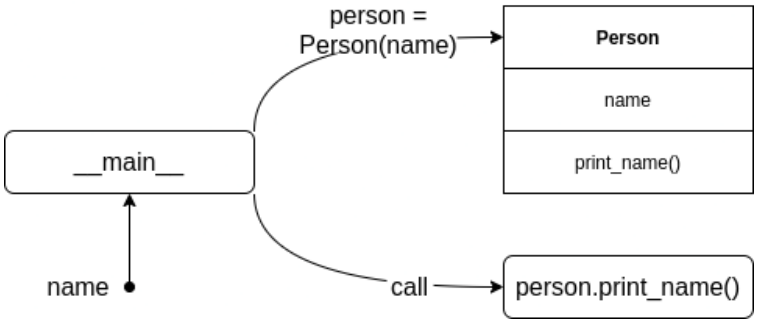
Figure A2: Example code in C to read in a number and check if it is even, using a user-defined function.

```c
#include <stdio.h>
int checkEven(int n);

int main() {
    int n1, isEven;
    printf("Enter a positive integer: ");
    scanf("%d", &n1);

    isEven = checkEven(n1);
    if (isEven == 1)
        printf("%d is even", n1);

    return 0;
}

// user-defined function to check if number is even
int checkEven(int n) {
    int isEven = 0
    if (n % 2 == 0) {
        isEven = 1;
    }
    return isEven;
}
```

Linkages between different pieces of software arise in a variety of ways:

- Composition creates dependencies between different classes; This is the primary channel through which linkages are created among different pieces of code;
- Implementation inheritance (as opposed to interface inheritance) creates dependencies between a parent class and its subclasses; A developer changing one of the parent classes might have to inspect all of the subclasses when making changes to ensure nothing breaks;

Figure A3: The "PrintName" program depicted as a network.



Notes: Python uses the special variable `__name__` and checks if it has the value `__main__`. If so, the code following is executed. In the example detailed in Figure A4, a person's name is read in and used to instantiate an object of the class `Person`. Then, that object's `print_name()` function is called.

Figure A4: Object-oriented example code in Python reading and printing a person's name.

```
# class definition
class Person:
    def __init__(self, name):
        self.name = name

    def print_name(self):
        print(self.name)

# main function
if __name__ == "__main__":
    name = input("Enter person's name: ")
    person = Person(name)
    person.print_name()
```

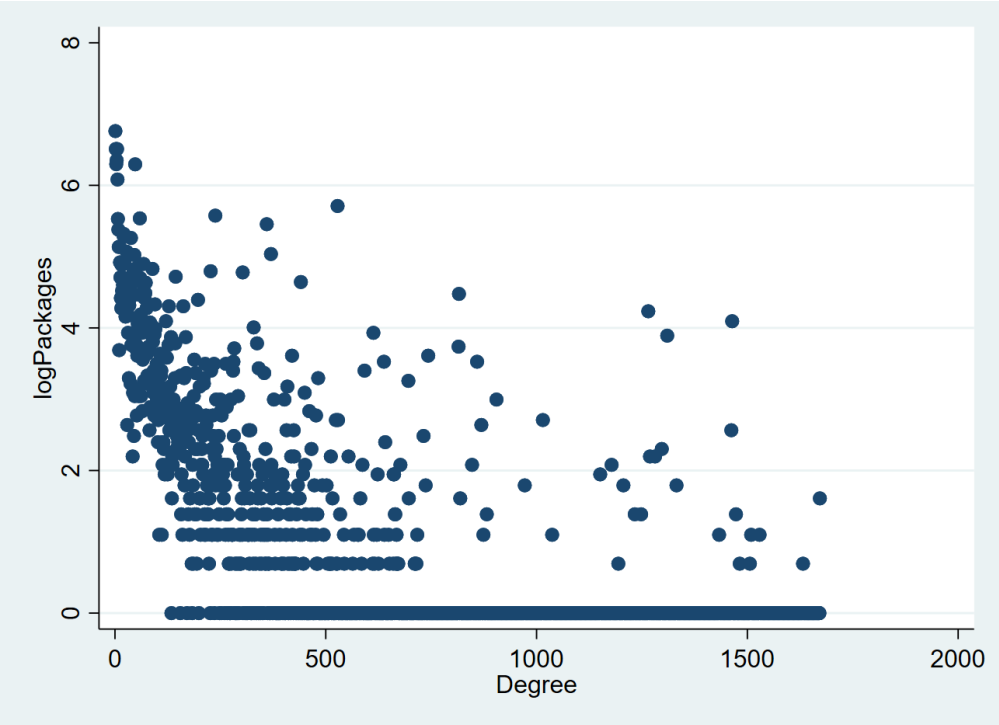
B Data processing

B.1 Rust dependency graph

We obtain the raw data from https://libraries.io, which includes three files: (i) A dependency file includes a project ID, a type of dependency (exact, inequality, wildcard), and the project ID of the dependency; (ii) A project file includes the project ID, a timestamp when the project was created and when it was updated, how many versions the project has, and a github repository ID; (iii) A version file which includes a project ID, a version number, and a timestamp when the version was published. Merged together, we obtain a file which includes the project ID, the github repository ID, a version number, and a dependency type as well as the dependency project ID. Dependencies of the inequality type imply that a dependency link will later be created in the dependency graph from the project to every version of the dependency that satisfies the inequality. Wildcard type dependencies imply that a link in the dependency graph will be formed from the project to every version of the dependency.

The merged dependency file has 599,972 dependencies of all types, which translate into a dependency graph with 17,081 nodes with 1,131,342 edges. The number of edges is larger than the number of dependencies in the merged file because of wildcard and inequality dependencies that are resolved when generating the dependency graph. Out of these, 16,525 nodes are connected by 1,128,963 edges in the largest weakly connected component. We restrict our sample on these packages. The degree distribution of the dependency graph is shown in Figure A5. The mean degree is 9.9 and the standard deviation 47.7. The distribution is skewed towards small values and over 80% of packages have no more than five dependencies (either to them from other packages, or to other packages).

Figure A5: Degree distribution for Rust's Cargo packet manager.



C Computational details for estimation

The likelihood of the model is shown in (9). Our scalable estimation algorithm exploits the fact that our model corresponds to a stochastic blockmodel when $\gamma = 0$, i.e. when there are no link externalities. More details are contained in Dahbura et al. (2021).

C.1 STEP 1: Approximate estimation of unobserved block structure

To recover the types and the block structure of the network, we approximate the model using a stochastic blockmodel. Define

$$L_0\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}\right) := p_\eta(\boldsymbol{z}) \pi(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\alpha}, \boldsymbol{\beta}, \gamma = 0) \tag{10}$$

This is the likelihood of a **stochastic blockmodel**, which means that

1. Each node belongs to one of K blocks/types;
2. Each link is conditionally independent, given the block structure

Then under some conditions – namely, that the network is large and each block/type is not too large with respect to the network (Babkin et al., 2020; Schweinberger, 2020; Schweinberger and Stewart, 2020) – we have

$$L\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\alpha}, \boldsymbol{\beta}, \gamma, \boldsymbol{\eta}\right) \approx L_0\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}\right) \tag{11}$$

Variational methods for the stochastic blockmodel are relatively standard and involve estimating an approximating distribution $q_\xi(\boldsymbol{z})$ that minimizes the Kulback-Leibler divergence from the true likelihood. This can be achieved in several ways, but usually the set of distributions is restricted to the ones that can be fully factorized, to simplify computations.

Formally, the full likelihood of our model can be written as follows

$$\mathscr{L}(\boldsymbol{g},\boldsymbol{x};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}) = \sum_{z\in\mathscr{Z}} L\left(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}\right) = \sum_{z\in\mathscr{Z}} p_{\boldsymbol{\eta}}\left(\boldsymbol{Z}=z\right)\pi(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\gamma). \tag{12}$$

where each firm's type is i.i.d. multinomial, so the distribution $p_{\boldsymbol{\eta}}(z)$ is

$$Z_i|\eta_1,...,\eta_K \overset{iid}{\sim} Multinomial\left(1;\eta_1,...\eta_K\right) \text{ for } i=1,...,n \tag{13}$$

Then the log-likelihood of our model can be lower bounded as follows. Let $q_\xi(z)$ be the auxiliary variational distribution that we want to use to approximate the distribution $p_\eta(z)$, then the log-likelihood has a lower-bound, calculated as follows,

$$\begin{aligned}
\ell(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}) &:= \log\sum_{z\in\mathscr{Z}} L\left(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}\right) && (14)\\
&\approx \log\sum_{z\in\mathscr{Z}} L_0\left(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right) && (15)\\
\text{multiply and divide by } q_\xi(z) \quad &= \log\sum_{z\in\mathscr{Z}} q_\xi(z)\frac{L_0\left(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right)}{q_\xi(z)} && (16)\\
\text{by Jensen's inequality} \quad &\geq \sum_{z\in\mathscr{Z}} q_\xi(z)\log\left[\frac{L_0\left(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right)}{q_\xi(z)}\right] && (17)\\
&= \sum_{z\in\mathscr{Z}} q_\xi(z)\log L_0\left(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right) - \sum_{z\in\mathscr{Z}} q_\xi(z)\log q_\xi(z) && (18)\\
&= \mathbb{E}_q \log L_0\left(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right) + \mathbb{H}(q) && (19)\\
&\equiv \ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi}). && (20)
\end{aligned}$$

where $\mathbb{H}(q)$ is the entropy of auxiliary distribution $q_\xi(z)$. The best lower bound is obtained by choosing $q_\xi(z)$ from the set of distributions $\mathscr{Q}$ that solves the following variational problem

$$\ell(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}) = \sup_{q\in\mathscr{Q}} \ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi}) \tag{21}$$

However, this variational problem is usually intractable, unless we impose more structure on the problem. In practice, researchers restrict the set $\mathscr{Q}$ to a smaller set of tractable distributions (Wainwright and Jordan, 2008; Mele and Zhu, forthcoming). In the case of the stochastic

blockmodel is useful and intuitive to restrict $\mathcal{Q}$ to be the set of multinomial distributions

$$\boldsymbol{Z}_i \overset{ind}{\sim} Multinomial\,(1;\xi_{i1},...\xi_{iK}) \text{ for } i=1,...,n \tag{22}$$

with $\boldsymbol{\xi}_i$ being the variational parameters. We collect the vectors of variational parameters in the matrix $\boldsymbol{\xi}$. This leads to a *tractable lower bound* that can be written in closed-form

$$\begin{aligned} \ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi}) &\equiv \sum_{\boldsymbol{z}\in\mathcal{Z}} q_{\boldsymbol{\xi}}(\boldsymbol{z}) \log\left[\frac{L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right)}{q_{\boldsymbol{\xi}}(\boldsymbol{z})}\right] &(23)\\ &= \sum_{i<j}^{n}\sum_{k=1}^{K}\sum_{l=1}^{K}\xi_{ik}\xi_{jl}\log\pi_{ij,kl}(g_{ij},\boldsymbol{x},\boldsymbol{z}) \\ &+ \sum_{i=1}^{n}\sum_{k=1}^{K}\xi_{ik}\left(\log\eta_k-\log\xi_{ik}\right) &(24)\end{aligned}$$

where the function $\pi_{ij,kl}$ is the conditional probability of a link between i and j of types k and l, respectively,

$$\begin{aligned}\log\pi_{ij,kl}(g_{ij},\boldsymbol{x},\boldsymbol{z}) &\equiv g_{ij}\log\left[\frac{\exp\left[u_{ij,kl}(\boldsymbol{\alpha},\boldsymbol{\beta})+u_{ji,lk}(\boldsymbol{\alpha},\boldsymbol{\beta})\right]}{1+\exp\left[u_{ij,kl}(\boldsymbol{\alpha},\boldsymbol{\beta})+u_{ji,lk}(\boldsymbol{\alpha},\boldsymbol{\beta})\right]}\right] \\ &+ (1-g_{ij})\log\left[\frac{1}{1+\exp\left[u_{ij,kl}(\boldsymbol{\alpha},\boldsymbol{\beta})+u_{ji,lk}(\boldsymbol{\alpha},\boldsymbol{\beta})\right]}\right]\end{aligned}$$

and

$$u_{ij,kl}(\boldsymbol{\alpha},\boldsymbol{\beta}) = u(\boldsymbol{x}_i,\boldsymbol{x}_j,z_{ik}=z_{jl}=1,\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta})$$

For very large networks the maximization of the lower bound (24) is still cumbersome and may be impractically slow. We thus borrow an idea from Vu et al. (2013) and find a minorizer $M\left(\boldsymbol{\xi};\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta},\boldsymbol{\xi}^{(s)}\right)$ for the lower bound. This consists of finding a function approximating the tractable lower bound $\ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi})$, but simpler to maximize. Such function $M\left(\boldsymbol{\xi};\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta},\boldsymbol{\xi}^{(s)}\right)$ minorizes $\ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi})$ at parameter $\boldsymbol{\xi}^{(s)}$ and iteration s of the variational EM algorithm if

$$\begin{aligned} M\left(\boldsymbol{\xi};\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta},\boldsymbol{\xi}^{(s)}\right) &\leq \ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi}) \quad \text{for all } \boldsymbol{\xi} &(25)\\ M\left(\boldsymbol{\xi}^{(s)};\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta},\boldsymbol{\xi}^{(s)}\right) &= \ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi}^{(s)}) &(26)\end{aligned}$$

where $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}$ and $\boldsymbol{\xi}^{(s)}$ are fixed. Maximizing the function M guarantees that the lower bound does not decreases. Vu et al. (2013) suggest to use the following expression

$$\begin{aligned} M\left(\boldsymbol{\xi}; \boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}, \boldsymbol{\xi}^{(s)}\right) &:= \sum_{i<j}^{n} \sum_{k=1}^{K} \sum_{l=1}^{K} \left(\xi_{ik}^2 \frac{\xi_{jl}^{(s)}}{2\xi_{ik}^{(s)}} + \xi_{jl}^2 \frac{\xi_{ik}^{(s)}}{2\xi_{jl}^{(s)}} \right) \log \pi_{ij;kl}^{(s)}(g_{ij}, \boldsymbol{x}, \boldsymbol{z}) \\ &+ \sum_{i=1}^{n} \sum_{k=1}^{K} \xi_{ik} \left(\log \eta_k^{(s)} - \log \xi_{ik}^{(s)} - \frac{\xi_{ik}}{\xi_{ik}^{(s)}} + 1 \right). \end{aligned} \tag{27}$$

Taking first order conditions with respect to each parameter implies the following closed-form update rules for $\boldsymbol{\xi}$, $\boldsymbol{\eta}$, and $\pi_{ij;kl}(g_{ij}, \boldsymbol{x}, \boldsymbol{z})$ follow

$$\boldsymbol{\xi}^{(s+1)} := \arg\max_{\boldsymbol{\xi}} M\left(\boldsymbol{\xi}; \boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}^{(s)}, \boldsymbol{\beta}^{(s)}, \boldsymbol{\eta}^{(s)}, \boldsymbol{\xi}^{(s)}\right),$$

$$\eta_k^{(s+1)} := \frac{1}{n} \sum_{i=1}^{n} \xi_{ik}^{(s+1)}, \quad k = 1, \ldots, K,$$

and

$$\pi_{ij;kl}^{(s+1)}(d, \chi_1, \ldots, \chi_p, \boldsymbol{z}) := \frac{\sum_{i=1}^{n} \sum_{j \neq i} \xi_{ik}^{(s+1)} \xi_{jl}^{(s+1)} \mathbf{1}\{g_{ij} = d, \chi_{1,ij} = \chi_1, \ldots, \chi_{p,ij} = \chi_p\}}{\sum_{i=1}^{n} \sum_{j \neq i} \xi_{ik}^{(s+1)} \xi_{jl}^{(s+1)} \mathbf{1}\{\chi_{1,ij} = \chi_1, \ldots, \chi_{p,ij} = \chi_p\}},$$

for $k, l = 1, \ldots, K$ and $d, \chi_1, \ldots, \chi_p \in \{0, 1\}$, respectively. The variables $\chi_{p,ij} = \mathbf{1}\{x_{ip} = x_{jp}\}$ are indicators for homophily. Generalizations are possible, as long as we maintain the discrete nature of the covariates $\boldsymbol{x}$. Including continuous covariates is definitely possible, but may result in a significant slow down of this algorithm. We note that when running the Variational EM (VEM) algorithm, we are not interested in estimation of the parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and γ, but only the estimation of probabilities $\pi_{ij;kl}^{(s+1)}(d, \chi_1, \ldots, \chi_p, \boldsymbol{z})$, a feature that allows us to speed up computation of several orders of magnitude (Dahbura et al., 2021).

We run this algorithm for many steps, and obtain the estimates for $\widehat{\boldsymbol{\xi}}$ and $\widehat{\boldsymbol{\eta}}$. We then assign each node to it modal estimated type, such that $\widehat{z}_{ik} = 1$ if $\widehat{\boldsymbol{\xi}}_{ik} \geq \widehat{\boldsymbol{\xi}}_{i\ell}$ for all $\ell \neq k$ and for all i's.

C.2 STEP 2: Estimation of structural utility parameters

In the second step, we condition on the approximate block structure estimated in the first step $\widehat{\boldsymbol{z}}$ and estimate the structural payoff parameters. We compute the conditional probability of a link within types and between types

$$p_{ij}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\gamma;\widehat{\boldsymbol{z}}) = \begin{cases} \Lambda\left(u_{ij}(\alpha_w,\beta_w)+u_{ji}(\alpha_w,\beta_w)+4\gamma\sum_{r\neq i,j} I_{ijr}g_{jr}g_{ir}\right) & \text{if } \widehat{z}_i=\widehat{z}_j \\ \Lambda\left(u_{ij}(\boldsymbol{\alpha},\boldsymbol{\beta})+u_{ji}(\boldsymbol{\alpha},\boldsymbol{\beta})\right) & \text{otherwise} \end{cases}$$

where $I_{ijr}=1$ if $\widehat{z}_i=\widehat{z}_j=\widehat{z}_r$ and $I_{ijr}=0$ otherwise; and $\Lambda(u)=e^u/(1+e^u)$ is the logistic function.

The **maximum pseudolikelihood estimator** (MPLE) solves the following maximization problem

$$\begin{aligned}(\widehat{\boldsymbol{\alpha}},\widehat{\boldsymbol{\beta}},\widehat{\gamma}) &= \arg\max_{\boldsymbol{\alpha},\boldsymbol{\beta},\gamma} \ell_{PL}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\gamma;\widehat{\boldsymbol{z}}) \\ &= \arg\max_{\boldsymbol{\alpha},\boldsymbol{\beta},\gamma} \sum_{i=1}^{n}\sum_{j>i}^{n}\left[g_{ij}\log p_{ij}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\gamma;\widehat{\boldsymbol{z}})+(1-g_{ij})\log(1-p_{ij}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\gamma;\widehat{\boldsymbol{z}}))\right]\end{aligned}$$

In practice the estimator maximizes the log of the product of conditional probabilities of linking. The asymptotic theory for this estimator is in Boucher and Mourifie (2017). It can be shown that the estimator is consistent and asymptotically normal. As long as the estimator for $\boldsymbol{z}$ provides consistent estimates, the estimator for the structural parameters is well-behaved.