Diffusion and Productivity of an Open Source Software: CakePHP

Master’s Thesis

Author:
Anwari Ilman
881206-6473

Supervisor:
Dr. Philippe Rouchy

Written in partial fulfillment of the requirements for obtaining the degree of Master of Science in Industrial Economics and Management (120 credits)

Specialization: Entrepreneurship and Innovation

Department of Industrial Economics
Blekinge Tekniska Högskola
Karlskrona, Sweden
January 2015
Abstract

The purpose of this thesis was to establish an understanding on the productivity of Open Source Software (OSS) developer community through diffusion of innovation. It was empirically ascertained that network effect affects productivity of OSS community, which provide enough justification to view the matter of productivity through the lens of diffusion of innovation. To reach its purpose, this thesis tackled the issue in two ways: 1) It utilized a definition of IT productivity for new technologies that especially deals with Open Source Software (OSS) communities and 2) It proposed tools and methods to perform such studies.

A mature OSS project called CakePHP was chosen as a case for this thesis. I compiled time-series data from the software’s source code that accounts for more than 8 years of development. The obtained raw data in a form of ‘commits’ was converted into network graph and time-series productivity data. Then, dynamic network visualization software was employed to analyze the evolution of its network structure. A quantitative regression analysis using Negative Binomial estimator was also employed to estimate the effects of individual work intensity, community work intensity and network effect on its production rate.

Visual inspection on CakePHP’s adoption pattern shows that it does indeed follow S-shaped diffusion curve normally found in other innovation life cycle, though yet to complete its life cycle. The regression results suggest that individual work intensity, network effect, and community work intensity were found to have significant effects on the rate of production output. The results also suggest that individual work intensity has a positive influence on the rate of production while network effect and community work intensity was found to have negative effects. This was suspected to be caused by overdispersion on the productivity level of contributors and also on the type of releases.

CakePHP underwent two diffusion phases, which are emergence phase and growth phase. These two phases exhibit very different network characteristics. On emergence phase the adoption rage, network size, number of connections, total production output and work intensity was substantially lower compared to growth phase. Members of the network especially opinion leaders and community leaders played a crucial role to drive adoption.
Acknowledgements

My profound gratitude goes to my mentor and supervisor Professor Philippe Rouchy for your professional and invaluable guidance from the process of idea proposal to eventually writing the final thesis. Your critical comments have shaped my independent theoretical and practical analysis. You gave me the opportunity to learn about research in Open Source Software, which is the key subject of my research. Please accept my sincere appreciation for your time and patience.

I acknowledge Sam Tavassoli, a PhD candidate at the department for your useful guidance and assistance on the subject of quantitative analysis during the process of writing this thesis. It has been a rewarding learning experience for me.

Greatest acknowledge to my beloved family and especially to my brother Ari Nurman who had given me enormous support through hard times during the writing of this thesis. To all my fellow peers and loved ones whom I met while living abroad, you all painted colorful memories to my life. It has truly been an amazing experience and I hope to take our learning and experiences into my future lives.

Lastly, to the Government and the people of Sweden, and especially the Swedish Institute, without financial support I would not be able to pursue my Masters education. I am profoundly grateful for the study scholarship.
Table of Contents

Abstract 1
Acknowledgements 2
Table of Contents 3
List of Tables 5
List of Figures 6
1 Introduction 8
  1.1 Background 8
  1.2 Problem discussion 8
    1.2.1 Problem of OSS productivity measurement 8
    1.2.2 Network effect as a determinant of OSS productivity 9
  1.3 Research questions and purpose 9
  1.4 Delimitations 10
  1.5 Thesis structure 10
2 Open Source Software, Productivity issues and Diffusion 11
  2.1 Open Source Software 11
    2.1.1 Definition of OSS 11
    2.1.2 Motivation to join OSS 12
    2.1.3 Research Challenges 13
  2.2 OSS as technical externalities 14
  2.3 Diffusion of Innovation 15
    2.3.1 Elements of diffusion 16
    2.3.2 Determinants of OSS diffusion 24
  2.4 The nexus of Productivity, Open Source Software and Diffusion 25
    2.4.1 Production output 25
    2.4.2 Contributing factors to production 26
3 Method 29
  3.1 Data 29
    3.1.1 Case selection 29
    3.1.2 OSS Community: CakePHP 31
    3.1.3 Data source 31
    3.1.4 Data issues 32
  3.2 Visualizing Network Structure 32
  3.3 Regression Model and Measurements 34
3.3.1 Dependent variable 36
3.3.2 Independent variables 37
3.3.3 Correlation matrix. 38

4 Result 39
4.1 Overall growth pattern 39
  4.1.1 Cumulative growth 39
  4.1.2 Active contributors growth 41
4.2 Regression findings 44
4.3 Productivity on emergence phase 45
4.4 Productivity on growth phase 46
4.5 Results on hypotheses 47

5 Analysis and Discussion 50
5.1 Determinants of diffusion 50
  5.1.1 Role of leaders 50
  5.1.2 Role of social structure 53
5.2 Evolution of diffusion network 55
  5.2.1 Diffusion on emergence phase 56
  5.2.2 Diffusion on growth phase 59
  5.2.3 Current state of diffusion 64
5.3 Implications of diffusion to productivity 66

6 Conclusions 68

References 70

Appendix 76
List of Tables

Table 1: Selection criteria of CakePHP .................................................................30
Table 2: Criteria of opinion leaders .................................................................34
Table 3: Variables and description of productivity .....................................37
Table 4: Correlation matrix and descriptive statistics ............................38
Table 5: Annual growth of contributors .....................................................40
Table 6: Regression result with Incidence Rate Ratio (IRR) parameters for predictors, time-series negative binomial estimation ..........................44
Table 7: Hypothesis testing result .................................................................49
Table 8: Opinion leaders in CakePHP ..........................................................51
Table 9: Annual growth of commits ............................................................76
Table 10: Regression result with raw coefficient parameters ................76
List of Figures

Figure 1: The order of file modifications (left) and its respective network model (right) ........33
Figure 2: Distribution of lines of code. The dependent variable is count data.........................36
Figure 3: Cumulative contributors of CakePHP. Growth phase started from around August 2009 ........................................................................................................................................39
Figure 4: New contributors (blue) and active contributors (black) per year. Year 9 is not a full year cycle........................................................................................................................................40
Figure 5: Cumulative growth of contributors June 2005 - January 2014 .................................43
Figure 6: Active contributors and new contributors June 2005 - January 2014 ........................43
Figure 7: Distribution of number of commits made by each contributor .................................48
Figure 8: Network structure of opinion leaders under PhpNut .................................................52
Figure 9: Network structure of opinion leaders under Mark_story .......................................53
Figure 10: Distribution of new contributors by roles .................................................................54
Figure 11: Dynamic visualization of CakePHP diffusion network can be accessed at Youtube. ...........................................................................................................................................55
Figure 12: Cumulative network in Year 1, by May 2006...........................................................56
Figure 13: Cumulative network by May 2007 .........................................................................56
Figure 14: Cumulative network by May 2008 .........................................................................57
Figure 15: Cumulative network by May 2009 .........................................................................57
Figure 16: Cumulative network by May 2010 .........................................................................60
Figure 17: Cumulative network by May 2011 .........................................................................61
Figure 18: Cumulative network by May 2012 .........................................................................62
Figure 19: Cumulative network by May 2013 .........................................................................63
Figure 20: Cumulative network by January 2014 .................................................................64
Figure 21: Current network structure of CakePHP in January 2014 ........................................65
Figure 22: Cumulative commit growth June 2005 - January 2014 ...........................................77
Figure 23: New monthly commits June 2005 - January 2014 ..................................................77
Figure 24: Network in Year 1, mid-May 2005-2006 ...............................................................78
Figure 25: Network in Year 2, mid-May 2006-2007 ...............................................................78
Figure 26: Network in Year 3, mid-May 2007-2008 ...............................................................78
Figure 27: Network in Year 4, mid-May 2008-2009 ...............................................................79
Figure 28: Network in Year 5, mid-May 2009-2010 ...............................................................79
Figure 29: Network in Year 6, mid-May 2010-2011 .................................................................80
Figure 30: Network in Year 7, mid-May 2011-2012 .................................................................81
Figure 31: Network in Year 8, mid-May 2012-2013 .................................................................82
Figure 32: Network in Year 9, mid-May 2013 - mid-January 2014 .............................................83
1 Introduction

1.1 Background

The issue of productivity is classically defined in relation to input and output of production (Solow 1957; Abramovitz 1956). For several reasons, the problem of assessment of software in terms of input-output is rendered difficult especially since it deals with immaterial assets (Boehm 1981; Baetjer 1998). This demands greater understanding on how software as immaterial asset is produced especially in a voluntary innovation environment such as Open Source Software (OSS). This thesis discusses the issue of productivity of OSS by focusing on how input-output production of software is better studied by capturing its community as a whole, as a system of software production.

One of the challenges this thesis deals with is the lack of knowledge as well as a procedure to tackle the issue of IT productivity. There are macroeconomic works that has been done in this domain (Brynjolfsson & Saunders 2010) but there is little attempt to deal with microeconomic phenomenon of productivity of OSS.

1.2 Problem discussion

1.2.1 Problem of OSS productivity measurement

The impact of software on the economy and issues of aggregate investments and prices of software is difficult to assess (Brynjolfsson 1993; Brynjolfsson & Saunders 2010). A number of researchers observe the difficulties in measuring software productivity (for example Baetjer 1998; Boehm 1981; Brynjolfsson 1993; Rouchy 2011). This is due to the complex nature of the product that is noticeably different from that of tangible good and services. The cause have also been exemplified by deficiencies in measurement and methodological toolkit rather than mismanagement (Brynjolfsson 1993).

Open Source Software (OSS) is a special knowledge asset that strives on technical knowledge of an interconnected community which creates a positive externality considering the technological division of labor and the labor effort that go into its activities (Kieseppä 2002). The level of connectedness and diffusion of the knowledge of participant in the community is motivated by shear benevolent collaboration and social value considerations (Lerner & Tirole 2002). This defining character of OSS development sets it apart from the other type of software that are developed traditionally.

In measuring productivity in the OSS unlike in other sectors where the level of education of field of specialty and output of firms can be used, this may not be replicated in OSS. The labor efforts put into software production come from vastly differentiated backgrounds of engineers and computer scientists (Ghosh et al. 2002). Yet these professionals are limited in supply. What is more interesting is that these skillful talents are often do not get financial incentive for contributing to OSS (Lerner & Tirole 2002; Kieseppä 2002; Lakhani & Wolf 2005). With the lack of financial incentive, it is not possible to measure productivity
by merely looking at labor incentive as production input. It is then important to understand what incentives motivate these skillful talents of developers to contribute to selfless cause like OSS.

Indeed, both intrinsic and extrinsic motivations that motivate developers range from altruism to peer recognition (Lakhani & Wolf 2005; Lerner & Tirole 2002; Rossi 2004). Motivations such as to gain peer recognition, better job prospects, and reputation require a larger size of network for it to be effective (Lerner & Tirole 2002). As the value of OSS for developers depends on the size of network, this reflects a relationship that is known as network effect.

1.2.2 Network effect as a determinant of OSS productivity

An important determinant that is suspected to affect the decision of a software developer to contribute to an OSS project is the existence of network effect (Lerner & Tirole 2002). Network effect occurs when the value of being a member of a network increases with the size of the network (Easley & Kleinberg 2010; Liebowitz & Margolis 1994). The existence of network effect is evidently displayed in OSS developers’ community within which innovation is diffused. The value of an OSS to a developer depends not only on its direct utility but also on the number of other people who contribute to the project (Raymond 1999). As each addition of new developer result in a change on the total production output, I suspect that network effect determines the productivity of OSS community.

Implicably, measuring productivity of OSS takes into consideration the OSS production function as a function of network effect. Taking a cue from the work of Fritz Machlup, (1962) I look at the economic implications of the OSS community in terms of supply of labor and output. This will be done by measuring work intensity in the OSS community, which means measuring the benefits produced by its contributors. This is synonymous to labor productivity. The calculation of work intensity gives the marginal product of labor. This will also be informed by the work of Simon Kuznets (1962) on inventive activity in terms of the magnitude of OSS outputs in the community. Meanwhile this whole work of measurement of productivity is anchored on the work of Robert Solow (1957).

One of the most important determinants of diffusion of an innovation is the time from its first inception until it matures (Rogers 1995). Since building a large enough network of any social system takes time (Hall 2004; Rogers 1995), it would be interesting to examine its dynamics and how it affects productivity over a longitudinal period. It is then reasonable to frame the issue of OSS developers’ productivity from the point of view of diffusion of innovation. This demands an understanding on the relation between network externality, how it causes diffusion and how it ultimately affects productivity.

1.3 Research questions and purpose

The issue of productivity investigated in this thesis is broken down into two research questions:
1. How does diffusion affect the productivity of OSS developer? It demands to ask if the community’s diffusion pattern follows the familiar S-shaped curve.

2. If there is such productivity, to what extent does network effect and work intensity at both individual and community levels affect productivity?

The purpose of this study is to establish an understanding on the productivity of OSS developer community through diffusion of innovation.

Understanding the process of OSS diffusion and key determinants of OSS productivity are done via extensive review of existing literature. I generally follow the funnel approach starting from problem of measurement of software to diffusion of innovation in order to provide a better understanding how this research relates to a broader context. Based on the findings in literature, I formulate several hypotheses and propose a model to capture them.

In order to substantiate the theoretical model claims and verify these hypotheses, I apply a focused study on a case of OSS community for its whole ongoing life cycle. The analysis and discussions depicted in this work include higher-level thinking and broader conclusions.

1.4 Delimitations

This thesis findings, analyses and conclusions are anchored on whole population of an OSS community for the period of its whole lifetime. Depending on the case selected, the software life cycle of the selected case is an ongoing process. I apply selection criteria to increase the chance of selecting a representative case for this thesis. Nevertheless, we may observe an incomplete life cycle of the OSS community.

1.5 Thesis structure

Following this introductory chapter, this thesis is organized as follows: Chapter 2 presents theoretical background and hypotheses at the end of its chapter. Chapter 3 discusses the method to answer the research questions while its results regarding hypothesis testing are presented on Chapter 4. Chapter 5 deals with the analysis and discussion on productivity of the community in relation to its diffusion. We end the thesis in Chapter 6 where I reflect the result of this thesis to its objective.
2 Open Source Software, Productivity issues and Diffusion

As argued in the Introduction section that in order to better understand productivity in networked production environment such as OSS requires a discussion of network effect and diffusion of innovation. Therefore, I start the discussion on this section with the elaboration of Open Source Software on their unique characteristics and rules that govern them. Then I move on to discuss the ability to capture the productivity of OSS developers through a point of view as network externality. We then continue by examining how network externality facilitates diffusion of innovation. I do so by reviewing the literatures on diffusion specific to OSS projects. In the end of this chapter I discuss the nexus of productivity, diffusion, OSS and how the latter two affect the former.

2.1 Open Source Software

The history of Open Source Software dated back as early as 1960s, where a practice of sharing code especially in academic settings was commonplace (Lerner & Tirole 2002). Academic institutions such as Berkeley, MIT and central corporate research facilities (e.g. Bell Laboratories and Xerox Palo Alto Research Center) are central to the early development of some breakthrough early OSS such as Unix and C language. This was motivated due to the fact that it was very costly to buy computer systems at that time, therefore software developers need to find a better alternative to use on their own systems (Ritchie & Thompson 1974).

In the 1990s, OSS gained worldwide popularity when Linus Torvalds released an operating system called Linux and distributed it over the internet (Lerner & Tirole 2002; Raymond 1999). He did so by encouraging other developers to contribute their code through online bulletin boards. Since then, the number of users grew rapidly from one hundred after the first year to half a million in 1994. Linux became one of the most well known OSS project at that time.

2.1.1 Definition of OSS

OSS is a positive technical externality regarded as public good that can be obtained at no cost by anyone without any rivalry (Open Source Initiatives n.d.). Essentially, Free or Open Source Software has two defining characteristics that distinguish it with proprietary software and freeware. The first character is that OSS source code, that is the human-readable form of computer instructions, is made to be freely available. Interested parties can obtain the unaltered form of the full source code that made up a program instead of being given the resulting machine code (or binary code) commonly found in the distribution of proprietary software. This differentiates OSS with freeware or shareware that can be obtained at no cost but does not provide access to its source code. Second, the nature of OSS license allows for free modifications and free distributions (Open Source Initiatives n.d.; Rossi 2004; Stallman 2009). One may improve an OSS by writing her own modification upon the former code, resulting in ‘derivative works’ that can be distributed by her own. As such, this practice is
restricted within proprietary software and freeware or shareware. Hence, the two defining characteristics allow interested developers whose locations are geographically distributed to modify existing programs and redistribute them.

Based on the obligations imposed on redistribution rights, OSS license can be categorized into ‘permissive’ and ‘restrictive’ license (Open Source Initiatives n.d.). The former type applies no restrictions on the distribution of the derivative works so that one may use it in proprietary application. Licenses that allow this freedom include BSD licenses and MIT license. The latter type applies restrictions on the derivative works to ensure that the code remains open so that any derivative works remains as public goods. This view is shared by the activists of Free Software, who advocates that “…the users have the freedom to run, copy, distribute, study, change and improve the software.” (Free Software Foundation 2014, p.1) so that the whole community benefits (Stallman 2009). Regardless of the permission imposed on the derivative works, the definition of OSS in this study does not strictly adhere to the restrictive definition of Free Software but comprises permissive licenses as well.

2.1.2 Motivation to join OSS

The growing interest on OSS from economists’ point of view is driven mainly by the motivation and organization that underlies the development of OSS. For economists to have the thought of an individual to spend considerable time and effort only to share his invention without any financial incentives is obviously startling, as it defies the economic principle of self interest in social action (Rossi 2004). Yet, countless OSS projects not only survive but also triumph by relying on voluntarily contributions among distributed developers (for example: Apache, jQuery, chromium, etc). Several authors tried to capture this selfless-interest by studying both intrinsic and extrinsic motivations that drives software developers to contribute to OSS project (Lakhani & Wolf 2005; Lerner & Tirole 2002; Rossi 2004; Varian 2006).

Intrinsic motivation deals with the inherent satisfaction one feels in doing an activity. Ryan & Deci (2000, p.56) defines intrinsic motivation as “the doing of an activity for its inherent satisfactions rather than for some separable consequence. When intrinsically motivated a person is moved to act for the fun or challenge entailed rather than because of external prods, pressures, or rewards.” According to this definition, having fun or enjoying oneself is the main idea behind intrinsic motivations. The idea behind enjoyment of doing an activity has been long recognized by Csikszentmihalyi (1975) who argues that some activities are pursued for the sake of enjoyment of doing it, even without any economic benefits. Csikszentmihalyi (1990) further argues that people who pursue these enjoyments are experiencing a state of ‘flow’ while doing their activities. Flow is the feeling of deep enjoyment sensation and full concentration that people feel when they act with total involvement (Csikszentmihalyi 1991). Characteristics that accompany state of flow typically are feeling strong, alert, in effortless control, unselfconscious, and at the peak of their abilities. Both a sense of time and emotional problems seem to disappear, and there is an exhilarating feeling of transcendence.

There are instances reported about how software developers experience a state of flow when writing software (Bilgin 2008; Evin 2014; Giorgio 2013; Lan 2012; Parnin & Rugaber 2011; Psygrammer 2011; Rachum 2012; Spolsky 2000). A research on motivations of OSS
developers by Lakhani & Wolf (2005) found intrinsic motivation have a stronger effect on the contributions of developers contrasted to extrinsic motivations. This finding suggests that OSS developers rather enjoy contributing to an OSS project for the sake of the creative feeling they experience when they work on it.

In addition to creative feelings, other top motivators are extrinsic motivations. Extrinsic motivations deal with benefits accumulating to an individual, through monetary compensation in general. But OSS project often lacks monetary compensation, although it is not uncommon to find software developers who are paid to contribute to one. As financial incentive often missing in OSS development, Lerner and Tirole (2002) argue that extrinsic motivations in OSS project manifest as either immediate or delayed payoff. Lerner & Tirole (2002, p.212-213) articulated this as “a programmer participates in a project, whether commercial or F/OSS, only if she derives a net benefit (broadly defined) from engaging in the activity. The net benefit is equal to the immediate payoff (current benefits minus current costs) plus the delayed payoff (delayed benefits minus delayed costs).” When immediate and delayed benefits in a developer’s cost-benefit calculation are taken into account, then voluntarily contributions are not as startling as it first appears. Hal L. Varian (2006) articulates two extrinsic motivations to join OSS community: “While open-source software is a public good, there are many motivations for writing open source software, he added, including (at the edge) scratching a creative itch and demonstrating skill to one’s peers.” (Varian 2006). Varian’s argument reflects individual needs to be fulfilled as an immediate payoff and job market signaling as delayed payoff. Hence, extrinsic motivations do not disappear completely; they exist but they do not manifest as financial incentives. Notable examples of extrinsic motivations in OSS projects are reputation, peer recognition, advancement in future career, user needs, learning and performance improvement (Lerner & Tirole 2002; Raymond 1999). Indeed, Lakhani & Wolf (2005) have found extrinsic motivations to have strong effect on contributions of developers includes user need, intellectual stimulation and improving programming skills.

2.1.3 Research Challenges

The relatively new field of Open Source Software leaves much area to be explored, especially from economic perspective. What makes OSS special is that it works under its own rules, which could be subsumed under the concept of “externalities”. Based on the discussions presented in previous sub-section, it is apparent that the OSS does not operate under the same condition as any other economic entity. Those externalities are more motivated by enjoyment rather than monetary compensation, organized independently, very innovative, etc. These conditions require us to adapt much of economical views to conform the rules and conditions governing OSS project. Hence, one must take into account the aforementioned OSS characters when discussing its diffusion within software developers as its social system. On the next sub-section I explore OSS as a concept of externality, specifically as a technical externality.
2.2 OSS as technical externalities

A product is said to exhibit network effect when the value that a user derive from a product depends upon the numbers of others who use the product (Katz & Shapiro 1985; Liebowitz & Margolis 1994). This idea comprises goods of ranging fields such as computer software, automobile repair, video games and many other goods. A particular example that exhibits network effect is the field of communication technology: e.g the utility of a telephone is more valuable when there are more people connected by telephone line. Whenever consumers benefit or suffer from changes in the network, either positive or negative, that is the time when goods exhibit a network effect. External factors such as compatibility, brand familiarity, product information, status, services availability or the prices of related goods are affected when a change in a product occurs.

For example, when someone is connected through telephone line, the overall network size expands hence increases the value of being a member of the telephone network. The telephone network case is an example of positive network externality. On this type of network, actions of an individual affect the welfare of other individuals without a mutually-agreed compensation (Easley & Kleinberg 2010).

But sometimes, network externalities exert not only positive but also negative effects to its consumers (Liebowitz & Margolis 1994; Easley & Kleinberg 2010). For example if a telephone network becomes overloaded, consumers will not be able to use the network properly. On this case the telephone network system is said to display a negative externality. Negative externality may occur in a common property when the property being accessed or used is not regulated properly. True common property may break down due to its limited supply and improper regulation. Hence, it requires explicit or implicit regulation that all the group members understood regarding their rights and duties on resource extraction.

On OSS case, it is quite obvious that OSS products exhibit positive network externality. For example, as an OSS project mature, the size of its user base largely reflects its utility. A large user base would imply that there would be more documentation, more features, and more support available for that particular OSS project. Notable examples on OSS includes as Linux operating system, Apache server, and Mozilla web browser (Easley & Kleinberg 2010).

OSS is regarded as non-depleteable resource and can be replicated at almost no additional cost, hence it falls under the category of open-access property as public good (Bessen 2005). It also lacks of rivalry since anyone can obtain any OSS without excluding others from consuming it due to its licensing features. Hence, OSS is not found to display negative network effect. The increase of network size is expected to exert positive output as technical externality. We will further examine OSS’ as network externality by distinguishing between direct and indirect network effect in OSS project.

Direct network effect occurs when there is a concrete network (either physical or virtual) made available through a product that its value depends on its network size. Direct network effect is generated through direct effect of the number of customers to the product’s quality. An example would be telephone line or railroad connection. As the number of cities that are connected by railroad increases, the value of the railroad increases in accordance to the number of cities.
On the other hand, indirect network effect occurs when the value of a product increases as its complement product increases. Indirect network effect lacks direct physical relation such as in case where complementary goods become more plentiful and cheaper. It does not need any concrete network between its users to exhibit network externality. An example would be the value of operating system is determined not by the number of users using it but rather by the number of applications that support that particular operating system. In this case, applications that are developed in that particular operating system act as complementary products thus increase the operating system’s value.

OSS exhibits both direct and indirect network externality. Weber (2000) pointed out that a direct externality is involved in OSS: the existence of a large number of users, though not necessarily skilled in programming, will ease the debugging process. They will be easier to spot flaws in software when compared to just a few core developers working on it. This view is also shared by Eric Raymond in his influential The Cathedral and The Bazaar (Raymond 1999). Hence, the existence of a large user base alone will benefit OSS by increasing its value to them.

For the same reason, OSS will also benefit when the number of developers who help to write the software increases. Unlike common users, developers are the agents who are on the supply side of software production. They are able to improve the software with their high technical skills. If software developers choose to share their fixes, not only this will increase the direct utility of the software but also it will increase the number of contributors who are involved in the community. The addition of new contributors will definitely enhance the overall network value of the community. Therefore, the involvement of new software developers will greatly enhance the direct network effect.

Network externalities shows that there are gain of innovation that is done at the periphery of the demand and supply of goods. It is done at the personal level of supply and demand of invention incentives. One solution to capture and evaluate supply and demand of software production as an incentive to invent is to investigate its adoption rate (Machlup 1962; Rogers 1995). I propose that the concept of innovation diffusion is useful for this purpose.

2.3 Diffusion of Innovation

It is a widely known fact that getting a new idea to be adopted is often found to be very difficult for new entrepreneurs (Rogers 1995). Not all innovations, no matter how innovative it is are accepted or used by potential adopters. It takes time and effort for an innovation to spread among its potential adopters. Some of them are well accepted while some of them are less successful in doing so. Hence, this reason motivates the devoted research attention to the issue of diffusion of new idea, process or object.

In recent years there has been a dramatic increase of Open Source Software adoption in terms of numbers of projects, users and developers (Lerner & Tirole 2002). There are also cases where OSS product are more favored and even dominates their market, e.g. the Apache Web Server, Mozilla Firefox, MySQL, or Linux OS (Rossi 2004). Governments around the
world also have endorsed various F/OSS software especially Linux over commercial closed-source operating system such as Windows or Mac OSX. A more recent successful endorsement of OSS by government is in Munich, Germany, where the local authorities decided to replace all its IT systems with OSS solutions (Linux Voice 2014). All these examples signify the diffusion of OSS projects in various social systems.

Despite the immense success of some OSS projects, it is not uncommon to find many OSS projects remain in emergence phase and their adoption rate never took off (Rajagopalan et al. 2010). This may due to an argument that the diffusion of OSS, especially in community of developers, is rather determined by the rate of productivity due to the network effect. This is reflected in Raymond’s (1999) argument that says the more a project is released, the more people will join the cause. In innovation studies, network externality and diffusion are rather quite new subjects in relation to productivity issues (Bonaccorsi & Rossi 2003; Rajagopalan et al. 2010).

Rogers (1995) articulated diffusion of innovation as “the process by which an innovation is communicated through certain channels over time among the members of a social system.” (Rogers 1995, p.5). Some investigations regarding diffusion in the field of software innovation has been conducted, but a methodological problem limits the observation and tracing process (Rogers 1995). The difficulty is mainly motivated by unobservable nature of software as information asset in physical sense, as discussed in sub-section 1.2.1. As have been discussed on sub-section 2.2, OSS production may be considered as an externality thus it is possible to investigate its diffusion through measuring the size of its network.

Despite the limits imposed in software innovation, some researches tried to provide an empirical framework to approach the innovation-decision process on the adoption of software. Research from Fuller, Hardin, & Scott (2007) identifies the accelerators and inhibitors of virtual world technology adoption on individual-, group- and organization-level. Evidences from two diffusion studies in the context of OSS project showed its diffusion pattern differs with other field of study. Whitmore, Choi, & Arzrumtsyan (2009) investigated the diffusion of eMule using number of downloads to proxy for adoption rate by users. They found the rate of adoption of eMule follows S-curve and can be modeled using Bass adoption model instead of logistic model. Another study is from Rajagopalan, Deshmukh, & Deshmukh (2010) where they investigated the rate of diffusion of healthcare related OSS innovation. They found diffusion patterns of OSS project could be classified into four groups, where its incubation time varies between projects. The first group does not show any explicit takeoff. The second group displays a very low activity for a time period prior to take off, showing a delayed takeoff pattern. The third group experiences a takeoff after a steady but insignificant increase in its first adopters. The fourth group unveils a direct takeoff pattern since the first time the innovation is introduced. Unsurprisingly their findings are consistent with Whitmore et al. where they found the rate of innovation could be explained using Bass adoption model.

2.3.1 Elements of diffusion

There are four identifiable elements involved in the process of diffusion of innovation, namely the innovation itself, the communication channels it uses to be spread, time period
needed to diffuse, and the social system where it diffused (Rogers 1995). The four key elements of diffusion of innovation have been well studied in a string of literatures in various subjects. In the field of software, however, the four elements differ with other material goods since software has its own defining characteristics. In light of software specificities, therefore these key elements shall be reviewed in the following subsections. There is still lack of adequate number of evidences regarding diffusion in context of Open Source Software as compared to other fields.

2.3.1.1 Innovation

According to Schumpeterian perspective, innovation is defined as the creation of new value, namely: new products, new method of production, new markets, new organization and new supply materials (Schumpeter 1934). While Schumpeter’s definition provides a rather broad concept of innovation, Rogers (1995) emphasizes the value of newness over the inherent concepts. Rogers stated innovation as a manifestation of an idea, process or object new to the perception of an individual or other unit of adoption. This idea of newness depends on the perception of the user rather than the objective measure of elapsed time since its first discovery. Put simply, if an idea is new to the user then it is an innovation. This newness aspect is important for the diffusion of an idea as not everyone is exposed to an innovation at the same time, thus they decided to adopt at different time.

Open Source Software itself has rather been considered as a breakthrough innovation from a number of perspectives (Klincewicz 2005; West & Gallagher 2006; Lerner & Tirole 2002). Lerner & Tirole (2002) cited OSS development process as a reminiscent of ‘user-driven innovation’ seen in other industries. This view is shared by Klincewicz (2005) who analyzed 500 most active OSS projects from Sourceforge.net regarding the degree of product innovation in terms of originality and characteristics. His study categorized 64 projects as innovative products while the rest operate under continuous incremental improvement. OSS has also has been investigated in terms of organizational innovation. West & Gallagher (2006) investigate the patterns of open innovation within firms and how OSS helps firms stay innovative. The organizational structure of OSS lends itself as an important organizational innovation as it relies on collaborative effort in a distributed working environment (Lerner & Tirole 2002; Bellantuono, N. Pontrandolfo & Scozzi 2012). In regard to its source innovators, Open Source Software project is one successful example of community-driven innovation since anyone with technical computer ability could contribute to OSS.

On the individual level, Rogers (1995) argues that these characteristics of innovation are the most important analytical categories in determining the rate of adoption of an innovation, namely: 1) relative advantages, 2) compatibility, 3) complexity, 4) trialability and 5) observability.

Relative advantage of an innovation may manifests in economic, social-prestige, convenience, and satisfaction terms. The rate of adoption increases as the perceived relative advantage of an innovation increases. For an individual to choose OSS as her software of choice may be motivated by reasons that ranges from economical to philosophical perspectives. A stream of literatures have examined the motivation of potential adopters to engage in the development of OSS (Lakhani & Wolf 2005; West & Gallagher 2006). Another
argument on the advantages of OSS is linked to the problems experienced by individuals thus became general solutions (Lerner & Tirole 2002; Raymond 1999). In that case, individuals try to solve their own problems and upon solving she realized that the solution is applicable to many others. This was the case of how Sendmail, Apache, and Perl were born.

The more complex an innovation, the slower its adoption since it would increasingly be more difficult to learn and understand (Rogers 1995; Hall 2004). As founders of Unix, Ken Thompson and Dennis Ritchie are aware that simplicity is key to attract other developers to quickly adopt and develop Unix OS (Ritchie & Thompson 1974). Simplicity in Unix philosophy emphasizes on building short, simple, clear, modular, and extendable software that can be easily maintained and repurposed by other developers. Until today, Unix’s philosophy to keep software simple and stupid is found to be very beneficial and now inspires most OSS projects to follow the same course. Fuller et al., (2007) argued the complex nature of virtual technology to affect the ability of individual adopters to adapt to innovative virtual environment. Moreover, he further argued factors like computer self-efficacy, creative self-efficacy, and personal innovativeness account for the individual level adoption decision as they require high level of computer, creative, and innovative competency, respectively.

When an innovation can be experimented prior to its adoption, any uncertainties that underlie the new idea decrease as the potential adopter learns more information about the idea. This newly formed information will in turn increases the rate of adoption (Hall 2004; Rogers 1995). Companies like Ubuntu tried to lower the barrier of adopting its operating system, by letting users to try the OS without the need to be installed in their machine (Ubuntu n.d.). Another example would be from one of Raymond’s philosophy where he advocates releasing software as soon and as often as possible. This practice is sought to increase trialability so that everyone can have the latest and greatest version of the software.

2.3.1.2 Communication channel

As suggested in some literatures (Rogers 1995; Whitmore et al. 2009), innovation alone cannot disseminate itself beyond its inventors had it not properly communicated. Diffusion relies on communication where participants create and share information to reach mutual understanding. Software engineering field has incorporated new communication methods that are specific to communities and are taking up in software development in general (Beck et al. 2001).

A study on OSS diffusion stresses the importance of communication for an innovation especially in their emergence phase. Utilizing new communication channel such as marketing is found to be significant in driving the growth of OSS project more than natural growth through ‘word of mouth’, especially during early stage. Whitmore et al. (2009) seek to explain the diffusion of OSS based on the size of users as single-factor growth model, which are suggested by previous literatures in OSS field. They used a case of eMule, an open source file-sharing client. They applied single-factor growth model to test the adequacy of the model in order to explain the growth of eMule. It turned out that single-factor growth model cannot explain the growth exhibited by eMule. In light of the first test result, they tested a two-factor model growth and it turned out to be a perfect fit to model the growth of the OSS project. The result suggested that relying growth on social system alone is not dependable. Adding new
communication channel like marketing plays a more important part in diffusion especially in
the initial stage of development.

Standard communication channels used by OSS includes online medium such as real-
time messaging, mailing lists, version control system, issue tracker, and website (Robbins
2005; Fogel 2006). To some extent, successful OSS projects such as Apache, Angular, Ruby on
Rails, jQuery, etc also utilize offline communication channels such as conference. Here are
some explanations on each communication channel:

2.3.1.2.1 Real-time messaging

Real-time messaging is a communication channel that designed for quick discussion,
especially question & answer exchanges between numbers of people (Oikarinen & Reed
1993). The interactions in this medium are found to be more informal and involves short but
frequent exchange of messages between members of a group chat. Depending on its
persistence, messages sent in real-time messaging system may or may not be recorded for
future access. IRC, the most popular messaging application for developers to hang out
together on a channel, do not to keep records of the conversations happened in that channel.
Despite its inability to persistently record messages, IRC channels became popular among
developers because it enables anyone to join a discussion channel anonymously. Other
commercially available chat programs such as Campfire, Skype, or Yahoo Messenger
persistently records its messages although more restrictive on who can see the messages.

2.3.1.2.2 Mailing list

Mailing list, or group mail is a persistent communication channel for members of a
group to exchange plans, questions, decisions or ideas that are accessible via web interface
(Robbins 2005). The medium allows for writing long message as opposed to short ones
commonly displayed in real-time messaging. The interactions do not occur in real-time.
Mailing list uses one centralized group email address to deliver messages right to all
members’ inbox instead of sending them to group chats. Combined with persistent message
records, group email exchange becomes very effective to reach a large number of group
members hence become the most frequently used communication medium.

Mailing lists are not generally managed manually by hand but they are managed
using mailing lists management software. This is useful to help responsible parties to deliver
messages to higher traffic easily. The software is also useful to moderate unwanted spam
messages. Some mailing lists management software are able to store all of the messages in
their own server, thus useful to archive them for future needs. Numerous open source and
closed commercial applications are available to manage and to archive mailing lists. Open
source tools to manage mailing lists include Mailman (www.list.org), Smartlist
and Ezmlm (http://cr yp.to/ezmlm.html). On the other side, open source solutions to
archive mailing lists include MHonArc (www.mhonarc.org), Hypermail
(www.hypermail.org), Lurker (http://sourceforge.net/projects/lurker) and Procmail
(www.procmail.org).
2.3.1.2.3 Website

A website is commonly used as a centralised one-way information channel from the project out to the public where anyone has universal access with diverse use and preferences (Robbins 2005). Website of an OSS project generally features some news, updates, releases, documentation, download page, and contributions sections. Websites are useful for OSS project leaders to release official information regarding the project. Hence, the existence of a dedicated website will increase the credibility of that OSS project. An OSS project does not necessarily has to develop their own website to in order to communicate with the their community. They may utilize a number of ‘portals’ served to host a large number of OSS projects. Websites such as SourceForge (http://sourceforge.net), Github (http://github.com), and FreeCode (http://freecode.com) allow project owners to host their Open Source project using centralized project management website.

2.3.1.2.4 Version Control System

Version Control System (VCS) or revision control system is an integrated source code management system to organize, track and control changes in a project’s files (Robbins 2005). The use of VCS applies on the software’s source code, its documentation, and even its web pages. VCS records every single change to the source code made by each software developers from the beginning of the project until the very latest change, including experimental changes that may not be stable enough to be deployed in production environment. The use of VCS is essential for a project with multiple developers to keep track of the contributions made by each of the contributors. Authorization issues can also be managed automatically by VCS system.

By using VCS as a communication channel, contributors able to manage, deliver, and track software source code faster and easier. It also enables them to troubleshoot technical problems quickly and safely as VCS provides an ability to reverse a software in development to its prior versions.

It is also important to note that VCS enables developers to experiment on software by making a development ‘branch’. A branch is one designated development tree. This way, a developer may make modifications of a software source code in her own local branch without disturbing the main development branch. This is especially useful when the experiment goes differently than the one planned by the real project owners. VCS also helps to automatically attribute software code, so it is quick and easy to know who made which contributions.

Some most popular VCS to date are Git (http://git-scm.com), Subversion or SVN (https://subversion.apache.com), Mercurial (http://mercurial.selenic.com) and CVS (not to be confused with VCS) (http://nongnu.org/cvs). Developers have used those systems for many years and most of them are familiar with them.

Some useful functions of VCS are as follows (definitions from Gitref n.d.):

- **Commit**: Commit is a recorded change of software source code made by a single developer in a programming session. A commit marks a historical state of source code in software development where each change is recorded as a state of software at that given moment. Every commit from the
beginning of a project until its current state is recorded in a VCS and can be retrieved to any recorded state.

- **Log message**: Log message is a bit of commentary attached to each commit describing the nature and purpose of the commit. There are important message bridging the gap between high technical language and user-oriented language of features, bug fixes and project progress.

- **Code update**: To keep following the latest change of software, a developer may update her own local copy of the project. She may do an update by incorporating changes made to remote repository into her own local copy. It is not uncommon to find developers update their software code several times a day.

- **Repository**: Repository is a centralized software code storage system. Main changes to software are usually kept at a main repository. Some version control systems are centralised in a single repository while others are decentralised where each developers have their own local copy of the repository. With decentralized system like this (for example, GIT) developers are required to synchronize their local version to the main version. The version control system keeps track of dependencies of changes required by a repository.

- **Checkout**: Checkout is a process of obtaining a copy of branch from a repository. A checkout produces a directory tree called a “working copy”. In some version control system, each working copy is itself a repository. Any changes made to this working copy may be committed back to the original repository.

- **Working copy**: A developer’s private directory tree containing the project’s source code. It also contains metadata managed by the version control system (telling the working copy which repository it comes from).

- **Revision, change, changeset**: a revision is a new version of a particular file or directory. For example: if a project contains a revision 6 of the file F, the commits submitted would change file F and assign a new revision of version 7.

- **Diff**: Diff is a function to compare two text documents on their differences between each other. It shows which lines that were changed, and whether it was a deletion or addition. The function is useful to give detailed information on how much a document has been changed and into what.

- **Tag**: Tag is a label for a particular collection of files in specific versions. Usually used to keep snapshots of the projects that are interesting. A tag is made for each public release, so developer can obtain a collection of files from the version control system from it. Tag is helpful for developers to identify software versions.
• **Branch**: a copy of the project, under version control but isolated – so that changes can be made to the branch without affecting the main project. Branch is commonly used for experimental development. It is made so that developers may make further changes to their own branch while preventing to destabilise the project development. A branch can be used to stabilise new release: during the release process, developmental work will continue uninterrupted on the main branch of the repository while, on the release branch, no changes are done unless the release managers accepts them. This practice makes sure that release branch will always be stable no matter what happen in the development branch.

• **Merge**: Merge is a practice to combine two different branches into one. Merging goes from the main trunk to other branches or vice versa. Merge is also what the version control system is doing when 2 people have changed the same file in a non-overlapping ways. If the changes do not interfere with each other, the update from one person will also automatically integrate the other person’s changes. If those changes overlap each other, then a ‘conflict’ will be raised.

• **Conflict**: Conflict is a flag that is raised when someone try to merge two different branches with overlapped changes made in those branches. It is usually the result of two people changed same files on different branches. VCS automatically detects the conflict and notify the one who wants to merge both branches. She will need to resolve the conflicted files before she could merge them.

• **Lock**: Lock is way to declare an exclusive intent to change a particular file or directory.

2.3.1.2.5 **Issue tracker**

Another specialized form of communication channel for software developers is a bug tracker. Issue tracker is a software application that keeps track of reported software bugs in a software development project (Robbins 2005). A bug is referred to as unexpected technical fault caused by overlooked part of software code. As the number of bugs may abundant in volume, developers need a systematic method to deal with them. This is where issue tracker system useful. Issue tracker system enables developers to query a bug and its subsequent information. This enables a number of developers to focus collaborating with each other to fight bugs at the same time without having to worry about managing them. Typically, issue tracking system are integrated with other software management applications. By using issue tracker system hand in hand with VCS, developers are able to keep track of which bugs they are working on, coordinate with each other, plan release, and plan new features.

Many issue tracking systems usually permit a third party to submit any bugs they have found into the system without requiring them to be registered in the system. This type of communication medium not only enables but also encourages outside party to be involved
in making the software more stable. External involvement makes testing and bug-spotting activities easier to be done as when compared to being done internally (Raymond 1999). This communication activity largely drives the course of development of the OSS project.

Notable alternative OSS solutions for tracking bugs are Mantis Bug Tracker (http://www.mantisbt.org), Bugzilla (http://www.bugzilla.org) and Debian bug tracking system (http://www.debian.org/Bugs).

2.3.1.2.6 Offline conferences

Offline activities such as conferences are sometimes organized to enable contributors, users and enthusiasts alike to get together and share visions, plans, knowledge and latest updates on their interest. Conferences are very effective channel as it brings people together in a same place so they can communicate and network directly with each other. OSS community usually organizes conference when there is substantial number of interested parties in the community. This may be the least frequent activity of all as it only happens usually once or twice a year.

2.3.1.3 Time

Given the discussion on software as network externality, we come to recognize that time is an important dimension of software innovation in OSS (Rogers 1995; Hall 2004). The network effect of OSS does not occur instantly at the time when the project is first introduced. Rather, it takes quite some time for OSS project to prove itself valuable in the eyes of potential adopters. Only when the software is thought to be valuable, an OSS will exhibit substantial growth phase.

The time dimension involved in diffusion of an innovation concerns three main cases: 1) in the innovation-decision process, 2) in the innovativeness of the adopters, and 3) in the adoption rate, usually measured as the number of members that adopt the innovation in a given period. The innovation-decision process is the first process through which an adopter decides whether to adopt or reject the innovation. The innovativeness of an adopter refers to the degree to which the adopter adopts the innovation earlier relative to other adopters. Adopters who are engaged as first mover are found to have high development cost and elevated risk (Klincewicz 2005). The adoption rate is the relative speed with which members of a social system adopts an innovation.

2.3.1.4 Social system

A social system is a set of interrelated units that unites to solve problem in order to achieve a common goal (Rogers 1995). Members of a social system may be individuals, organizations, or subsystems. OSS traditionally serves two social systems within which innovation is adopted according to their functions, namely users and developers.

Users are a group of individuals who uses OSS to solve their problem. They are not necessarily educated with technical computer skills such as software engineering, computer science or programming. They mainly use the software to benefit from its functionality without bothering to experiment with its underlying technical part. On the other side,
developers are a group of individuals who are not only use the software but also can improve upon it. They have skills in computer science or programming and are willing to dedicate some time and effort to improve the software. Developers are also users, but users may not be developers. Most of studies on OSS focus the issue of diffusion on users as social system but not on developers (e.g. Rajagopalan et al., 2010; Whitmore et al., 2009).

Social system covers the concepts of social structure, norms, and roles of opinion leaders and change agents (Rogers 1995). A social structure is bound to take place for a social system to function properly. The structure gives regularity and stability thus allow researchers to predict behavior based on the role of its members. Thus, the existence of social structure represents a type of information that reduces uncertainty.

In a social system of developers, social structure is usually represented by roles of developers. Traditional software development model (the Cathedral) commonly involves several roles addressed as software architects, software managers and coder. Unlike the Cathedral, OSS development model (the Bazaar) often lack the formal social structure found in traditional software development (Raymond 1999). Nevertheless, a well-maintained OSS project usually has an informal hierarchy of roles (Kazman & Chen 2009; Gardler & Hanganu 2010), and at the least they distinguished between core contributors and ordinary contributors.

The behavior of one individual is found to affect the behavior of his fellows. It is common to find opinion leaders or change agents in a social system (Rogers 1995). Opinion leaders are members of a system who provide information and advice about the innovation to the other members (Rogers 1995). The position as opinion leaders usually earned and maintained by having technical competence, social accessibility, and conformity to the system’s norms. Change agents are individuals who influences clients’ innovation decision. They usually seek to either encourage or hinder the adoption of a new idea before an individual makes a decision to adopt it. The role of opinion leaders or change agents in OSS project may further be distinguished using several criteria. This study identifies practical criteria applied to filter the role of opinion leaders in chapter three.

2.3.2 Determinants of OSS diffusion

Understanding the process of diffusion requires a larger framework that deals on the society level. Rogers (1995) points external or social conditions that may accelerate or slow the process of diffusion: 1) The agent who made decision to adopt whether it is individual, collective or an authority, 2) The communication channel, 3) The nature of its social system, norms, and degree of interconnectedness and 4) The extent of change agents’ promotion effort. The first and third social conditions directly deals with how innovation diffused in a social systems. We have discussed the role of social systems in which OSS is diffused in sub-section 2.3.1.4. The second and fourth social conditions deal with how information is communicated with various communication channels, as we have discussed in section 2.3.1.2.

Combined with the characters that drive the decision to adopt on the individual level, these external or social conditions are then translated by Hall (2005) into determinants of innovation namely 1) benefits received, 2) network effect, 3) cost of adoption, 4) information
and uncertainty, and 5) market size, industry environment and market structure. A number of literatures have also dealt with these issues related to OSS projects.

There are studies on software innovation that tried to provide a framework to observe factors that affect diffusion rate. Fuller et al., (2007) studied innovation-decision process of ‘virtual innovation’ and argued that on different stages, agents such as specific individual, group and organizational level factors serve as accelerator or inhibitor of adoption.

A number of studies characterized network effect as a critical factor in the diffusion of OSS and software in particular (Bonaccorsi & Rossi 2003; Rajagopalan et al. 2010; Whitmore et al. 2009). The role of network effect in the diffusion of OSS is suggested to be more emphasized especially in environments that are dominated by proprietary software (Whitmore et al. 2009).

2.4 The nexus of Productivity, Open Source Software and Diffusion

As a software technology, OSS is regarded as a breakthrough technological innovation in terms of its development process (Bonaccorsi & Rossi 2003). However, the activities carried out by OSS developers are principally similar to those who work in traditional software development (Stallman 2009). The activities such as writing codes and commit it are needed to improve the overall software. When the written code meets its pre-determined release goals then the core team releases newly revised software and label it as a new version (Raymond 1999).

The fact that OSS is essentially software that only differs in the way it is developed from traditional software means that traditional measure of productivity used in software engineering is applicable to OSS projects. We will look at how productivity measure is approached in OSS project.

2.4.1 Production output

Like any other software technology, OSS suffers from some degree of technological uncertainties (Lehman & Ramil 2002). This uncertainty aspect applies differently to individuals depending on which role uses the an innovation (Rogers 1995). For the users, uncertainties traditionally arise around whether the OSS in question has sufficient support, sufficient user base, and if it is stable enough to be used in production setting. For the developers, however, uncertainties arise when the technology is not yet stable. Unstable software may not function properly as expected, so that a developer who uses it may need to fix the software herself.

This technological uncertainty problem led to a pretty straightforward solution, which is by fixing any bugs that may arise and also by refactor existing code or enhance some features. The result of this software development activity would be an increase (and sometimes decrease) in number of lines of code. Writing more code implies an effort to develop software thus increases its benefits for potential adopters. This will in turn reduces uncertainty in the eye of potential adopters. Therefore, measuring lines of code could be used to approximate both productivity output and technological uncertainty as well. Counting
lines of code also commonly considered to be a widely-used measure on the output of programming activity (Ramirez & Nembhard 2004).

String of literatures that deals with the issue of productivity of software has stemmed from the field of Software Engineering (Boehm 1981). One of the most widely used measurements of software cost is the COCOMO model (Constractive COst MOdel) proposed by Dr. Barry Boehm (Boehm 1981; Boehm n.d.). There are some variations on the COCOMO models depending on the assessment on the project. At the basic level, COCOMO model measures software development effort based on program size (number of lines of code).

There has been extensive research and practices carried out in regard to counting lines of code (loc) to measure the productivity of software worker (Jones 1978). Of course, the method of counting loc as the oldest and most straightforward method has its own advantages and drawbacks. One of the most cited drawback is that counting loc only account for quantity and simply ignores the quality of code (Jones 1978). Productivity then means a simple matter of writing more lines of code, not writing better, useful code. Nevertheless, software size provides a good estimation of how much effort is produced into developing the software. This will then will inform how much the software cost.

On another side, counting loc provides a relatively straightforward procedure to measure software developer productivity compared to other methods such as Function Points Analysis (FPA) (Ramirez & Nembhard 2004). In counting lines of code, an analyst only needs to calculate the number of lines of code that are produced. Measurement methods such as FPA require productivity analysts to assign points to each functions of the software. FPA is usually measured by the ratio of function points (FPs) delivered to programming hours and months. However, the rating of the raw count in FPA ultimately is subjective as its accuracy depends on the skill and experience level of the analyst who performs the counting (Ramirez & Nembhard 2004).

Nevertheless, FPA also exhibits problems and issues when measuring software engineering productivity. Three factors are critical to a successful implementation of an FPA method, namely the analyst’s knowledge of FPA, calibration of the function point productivity indicator, and rigor of the measurement process (Bok & Raman 2000). Considering the inherent difficulty that accompanies FPA analysis, this thesis uses counting lines of code as its measure for production output.

2.4.2 Contributing factors to production

2.4.2.1 Individual’s work intensity

As has been discussed in sub-section 1.2.1, software is difficult to measure due to the complex nature of the product that is noticeably different from that of tangible good and services. The labor incentive of working, as a channel of capital input, is one defining factor in traditional productivity measurement that defines production function (Solow 1957). Unfortunately, in OSS projects labor incentive is not something that can be measured since most of the developers work voluntarily without any financial incentive. Clearly, the lack of financial incentives suggests that there are other forms of incentives to compensate developers for their effort and extra time spent to work on OSS projects.
As has been discussed in sub-section 2.1.2, one type of incentive OSS developers obtain from contributing to OSS project is the immediate payoff they gain by modifying the software code. Developers in particular have strong incentives to create custom solutions to their particular needs (Lakhani & Wolf 2005). Another type of incentive would arguably be the satisfaction and creative feeling developers acquire from the fulfillment of their intrinsic motivation. Developers experienced the feeling of creativeness when they write software code. This process of fulfilling intrinsic motivation is then argued to exhibit a state of flow, as has been reported in numerous cases (for example Bilgin 2008; Evin 2014; Giorgio 2013; Lan 2012; Parnin & Rugaber 2011; Psygrammer 2011; Rachum 2012; Spolsky 2000).

OSS developers experience a state of flow when they write and consequently finish software code by committing it (Bilgin 2008; Giorgio 2013; Rachum 2012; Psygrammer 2011; Spolsky 2000). Writing and committing codes do not happen in one long session but rather on small and short period as batches (Gitref n.d.). As they work on batch, software developers add a few lines of code as a refinement of the final product. Each of these batches constitutes a commit, where a commit can be defined as single session of coding consisting an updated part of software. Being in state of flow stimulates developers to keep making commits, creating work traction to keep writing and submitting code until they are satisfied of what they write in a programming session. We refer to this part as work traction.

Since early 1950s, work traction is argued to have a significant effect on the output produced (Baldamus 1951). One type of this effort that keeps a worker to keep working is object traction, which described as “having mental picture of the work-object or its parts, which one feels urged to reproduce.” (Baldamus, 1951:p48). This kind of traction is particularly apparent in highly skilled work, which requires mental process to ‘complete’ a batch or a unit of job. As the completion of the job is approaching, the intensity of object traction increases. Hence, object traction to keep working is commonly exhibited in most software development activities especially when writing programs.

Since labor incentive in the context of OSS does not manifest as financial incentive like in other economic activities, therefore we need to readjust its meanings to the context of how OSS works. This concept should encompass psychological incentives such as intrinsic and extrinsic motivations. Based on the discussion above regarding OSS’s characteristics, it is plausible to say that, labor incentive in OSS may be represented as traction effort, intrinsic motivation, as well as its immediate payoff. Considering that those terms are closely related with work intensity on individual level, they are then subsumed as ‘individual work intensity’.

Moreover, as labor incentive is argued to affect production output linearly, therefore individual work intensity is expected to exhibit similar result. It is suspected that the increase in individual work intensity will result in the increase of production output. Therefore Hypothesis 1 is as follows:

*Hypothesis 1: Individual work intensity has a positive and significant effect on the production output of OSS*
2.4.2.2 Network effect

Another factor that is argued to have significant effect on production output is network effect. As has been discussed in sub-section 2.2, software products in general are known to exhibit technical externality. This also holds true for OSS considering its production take place in a networked environment.

The existence of network externality is argued to increase OSS productivity since the more people join, more effort will be put to produce the software (Raymond 1999; Lerner & Tirole 2002). This holds true to some extent, as each new addition of contributor would increase production on the aggregate level. But we shall not dismiss the fact that not everyone has equal skill, chance or commitment to contribute to an OSS project. Past research from Ghosh & Prakash (2000) indicates an existence of enormous gap of software produced between the most productive and the least productive contributors. The result of the research shows that productivity depends largely on skills and commitment of the contributors. If network effect attracts low-committed contributors more than high-committed ones, then it may result negatively on the average productivity. Despite the level of commitment of contributors, we anticipate network effect would affect production output significantly.

The diffusion of OSS among developers does not occur instantly at the time when the product was firstly introduced. Rather, time is of key importance here. Innovation-decision process, innovativeness of adopter, and adoption rate determine the time when an innovation would show signs of network effect. Adoption process is known to grow slowly until a number of users form a ‘critical mass’. By that time, network effect will be more evident (Rogers 1995). Hence, it is suspected that network effect will be more apparent at a later stage than early stage of development.

As network effect is expected to affect production output significantly, therefore Hypothesis 2 is as follows:

**Hypothesis 2: Network effect has a significant influence on the production output of OSS**

2.4.2.3 Community’s work intensity

As the completion of a job is approaching, the intensity of object traction increases as well (Baldamus 1951). This phenomenon may be observed in OSS community when a planned release date is approaching. Prior to a release date, motivated developers may feel compelled to work harder to complete the milestone on that version. Hence we suspect traction intensity would increase, as dedicated developers shall try to fix bugs or add new features to follow their scheduled release. This continuous refinement will arguably be reflected by an increase of production output. The aggregate individual effort to meet the deadline is expected to be displayed as the community’s traction should the whole team is dedicated enough to exert more effort prior to the release date. Hence, I suspect object traction not only affects productivity at individual level, but also at community level. Hypothesis 3 is as follows:

**Hypothesis 3: The community’s work intensity has a significant and positive influence on the production output of OSS**
3 Method

Research methods generally divided into two main approaches, which are quantitative or qualitative research (Myers 1997). Qualitative research is a situated activity that locates the observer in the world. It involves the use of qualitative data sources such as interviews, observations, discussion, recordings, etc (Ghauri & Grønhaug 2005). It tries to develop theories inductively where its main concern is to look for meanings rather than patterns or behavior (Silverman 2013). On the other hand, quantitative research is a deductive process of obtaining knowledge through theory testing (Ghauri & Grønhaug 2005). It is particularly suitable to address research questions asking what or how many. This approach is a formal systematic process in which numerical data findings are statistically analyzed in order to describe, test and examine cause-and-effect relationships.

Considering the research questions asked in this thesis deal with recognizing diffusion pattern and testing hypotheses that contains numerical change, then quantitative method is considered appropriate to answer them. The first question regarding the diffusion pattern is approached with descriptive explanation on its rate of adoption through the overall ongoing project’s lifetime. On answering this I use network structure visualization, since it gives helpful insight for examining patterns and represent network compactly (Hanneman & Riddle 2005; Moody et al. 2005). The second question regarding the extent to which individual network effect, work intensity, and community’s work intensity account for the rate of production will be answered using regression analysis. Indeed, regression analysis is suitable to estimate the relationships between variables and to test for hypothesis testing (Ghauri & Grønhaug 2005). I outline the regression model on sub-section 3.3.

As suggested by Rogers (1995), a particularly robust diffusion research inquiry would investigate the issue of diffusions at two or more points during its diffusion process rather than after its completion. Based on this ground, I later divide the analysis into periods according to innovation phases, each of which has distinguishing characteristics compared to the others. The answer for the second question will be adjusted to each phase accordingly.

3.1 Data

3.1.1 Case selection

This thesis uses a case study of an OSS community as its analytical unit to help frame the boundary of this study. The motivation behind the use of case study was because case studies provide empirical data from real setting, necessary to describe a particular phenomenon (Yin 2003). This reason provides a firm confidence to use a case study to appropriately answer this thesis’s research questions.

A theoretical sampling strategy was employed in this study where we select a case based on their expected contributions to theory development rather than for representativeness. To get a relevant case, I employed general OSS relevancy criteria adopted in Crowston, Li, Wei, Eseryel, & Howison (2007) where they applied similar criteria in their study. First, the source code repository of the project had to be publicly available (excluding those projects which has their repository stored in personal cloud or require registration to be
accessed). Second, the project consists of more than 50 contributors to provide more reliable number of observations. Higher number of developers also matters because small projects are less likely to show recognizable pattern of diffusion. Third, the project had to be relatively successful at managing the contributions of multiple developers (core developers and many more peripheral contributors).

The project was also assessed using OSS project’s success criteria (Crowston et al. 2003), which are: have attracted numerous developers beyond initial founders, are continuing to release software, have extensive documentation and have an active user community that provides feedback.

According to these criteria, an OSS project called CakePHP is then selected as the case for this study as it satisfies both general and success criteria previously discussed. It is specifically chosen since it is one of the first web development frameworks that have been introduced. The project has existed for almost 9 years and has hundreds of contributors. With long period of development and many contributors, the project is expected to provide more observation than the other projects in the same category. This thesis uses the whole population of CakePHP developers as its representative data. The selection criteria of CakePHP are presented in Table 1.

Table 1: Selection criteria of CakePHP

<table>
<thead>
<tr>
<th>No</th>
<th>Selection Criteria</th>
<th>CakePHP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>General Criteria</strong> (Crowston et al. 2007)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Accessibility of source code</td>
<td>Publicly available</td>
</tr>
<tr>
<td>2</td>
<td>Number of contributors</td>
<td>270 (as of January 15, 2014)</td>
</tr>
<tr>
<td>3</td>
<td>Development status</td>
<td>Active</td>
</tr>
<tr>
<td>4</td>
<td>Latest Release</td>
<td>2.4.4</td>
</tr>
<tr>
<td>5</td>
<td>Actively accepting contributions</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td><strong>Success Criteria</strong> (Crowston et al. 2003)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Has contributed</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Release cycle *</td>
<td>Major release every 3 years</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor release every 4-6 months.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Patch release every few weeks.</td>
</tr>
<tr>
<td>3</td>
<td>Has extensive documentation</td>
<td>Yes, accessible via <a href="http://book.cakephp.org">http://book.cakephp.org</a></td>
</tr>
<tr>
<td>4</td>
<td>Has active user community</td>
<td>Yes, through <a href="http://community.cakephp.org">http://community.cakephp.org</a></td>
</tr>
</tbody>
</table>

*CakePHP follows semantic versioning system ([semver.org](http://semver.org))*
The following sub-section introduces CakePHP project with descriptive explanation about its history and its current status.

3.1.2 OSS Community: CakePHP

CakePHP is a web development framework in PHP language that is designed to enhance the productivity of developers by abstracting most used programming functions (CakePHP Foundation n.d.; Github 2013; Watts 2013). On April 15th 2005, Michal Tatarynovicz (a.k.a. Pies) released a minimal version of rapid application framework in PHP and called it ‘Cake’. The objective of this community is to provide technical tools for helping PHP developers to write web application faster, simpler and easier (CakePHP Foundation n.d.). It provides extensible architecture for developing, maintaining and deploying web applications. Using commonly known design patterns, it reduces development costs and helps developers to write less code. The tool was inspired by the success of Ruby on Rails, a similar web development tool written in Ruby language.

The newly released project immediately drew developers’ attentions and some even engaged to contribute to the main source code. Just a little over a year since the project was made public, a stable CakePHP version 1.0.0 was released on May 1st 2006. Over the course of 8.5 years, the project has seen a faster release cycle, attracted new contributors, organized annual conferences (known as ‘Cakefest’) and gain new sponsorships. There were also some structural changes happened in the community when lead developers chose to abandon the project and was overtaken by someone new.

Although the development was initiated in 2005, it was not until mid-2008 CakePHP gained massive attention from developers. Starting from 2008, the community grew faster with more than 10,000 likes and followers on Facebook interest page and twitter. The official group on Facebook surpassed 3,000 members, larger than groups for other major PHP frameworks. The growth in terms of number of visits to CakePHP’s official website grew for more than 1 million visits compared to 2012. The visits on documentation page reached over 26 million views (Watts 2013). Until January 2014, the CakePHP community has released version 2.4.7 and is actively developing a new major version of CakePHP 3.0.* to be released in 2014.

3.1.3 Data source

The primary dataset used for the study were contributions to the main source code, which are managed through a source code Version Control System (VCS). A commit is defined as a recorded compilation of modified source code made by a programmer in a single session (Gitref n.d.). The commits recorded in VCS contain very detailed information such the person who made the change, when the change was made and the description of the change. Commits even contain information about the files that were affected and code that were added, modified or deleted in a particular commit. Every single commit can be identified with an ID to differentiate it with another commit, so they can be searched, identified and tracked easily based on their ID. Commit also contains information about its parent(s) and consequently its preceding commits before them. This allows developers to trace the ancestors of a particular commit up to its origin.
For the purpose of this thesis, I compiled the source code of CakePHP that covers its whole development period up until the time of data collection for this thesis. It accounts for more than 16 thousand commits for almost 9 years from May 2005 until January 2014. Over this course of 104 months, there were in total 273 developers contributed to the repository. The repository accounts for 4784 files with more than 100,000 changes made upon them. This dataset was captured from the official CakePHP source code repository at Github.com, which is freely available to be accessed via Git. In addition to technical data, this study also employed historical data obtained from the official website (http://cakephp.org), mailing lists and developers blog.

3.1.4 Data issues

It should be noted that the commits data on this study undergone a conversion of VCS system. Prior of using Git, CakePHP used Subversion (SVN) as its main version control system. The conversion affects committing practice in the operational level and requires the data from SVN to be extracted into Git. This change caused some data incompatibility between those two versioning systems. It comprises ‘tagging’ feature available in Git, therefore there were no complete tagging information prior to CakePHP version 1.2.0. This will potentially cause missing values on our planned regression. To compensate this, I use historical data through mailing lists messages and official website’s blog posts to look for release dates.

Another issue with the data would be a duplication of individuals. To identify individuals who are involved in writing CakePHP, I extract their data from unique email address registered in CakePHP’s Git. The raw dataset freshly pulled from Git repository contains of more than 320 unique emails. A number of these unique emails turned out to be used by the same person where a similar name is associated with more than one email. This practice is quite common considering some developers may work from different computer machines, as they need to identify each machine with unique email. This is useful to distinguish which commits were made in which machine. To avoid duplication of developers, I remove unnecessary extra emails and assign a developer with a single identifiable email address.

Some modified files that are recorded may also have technical issue. Some files are marked as ‘modified’ although neither additions nor deletions were made to the files. This may occur during a ‘merge’, that is a combination of two branches into one. In the resulting branch, the merged files may be left unmodified hence no addition or deletion is detected for that particular file. When no modifications were found upon the merged files, they are not included in the measurement of lines of code.

3.2 Visualizing Network Structure

Dynamic network visualizations is a useful tool to foster theoretical insight in a longitudinal social networks (Moody et al. 2005). It gives helpful insight as to how actors behave in a network. On this thesis, visualization is important because it is a way to present productivity in a community that may not be visible when analyzed solely using statistical process.
To display how social structure evolves based on time-series data, the commits data was converted into network graph data. A graph is composed of nodes connected by edges and may represent a single type of relations among the actors (simplex), or more than one kind of relation (multiplex) (Hanneman & Riddle 2005). This study employs dynamic network visualization as a tool to visualize its network structure.

Using graph to represent social relation is anchored on the work of Hanneman & Riddle (Hanneman & Riddle 2005). This thesis models the network of diffusion based on the modification of files. It traces the order of file modifications by listing contributors who made modifications on that file with the information found in a commit. The order of file modifications is of key importance because it determines the direction where the diffusion headed. When a file is written by programmer A and then modified by programmer B, it is said that B adopted the software through A.

In this network model, the term ‘actor’ refers to someone who made modifications to a file and commit it. She is then modeled as a node in the visualized network graph. To form a network, the nodes are connected to each other with a relationship. In our case, the relationship is formed when someone decided to modify someone else’s work. This relationship hence depicted as edge. The edge that connects two nodes can be distinguished as directed edge. A directed edge is represented as arrow in the network so that if B adopts the software from A can be depicted as B\(\rightarrow\)A. Figure 1 shows the process of modeling a network of developers.

Figure 1: The order of file modifications (left) and its respective network model (right)

Visualization software named Gephi is employed to dynamically visualize the network of CakePHP. In the visualization result, size of the node represents number of connections an actor has to other actors. The bigger the size of the node, the more connected that actor hence the more central she is. The rendered network is also have been set to push more central actors to the middle of the network while less connected actors are pulled towards the outside of the network. This makes it easier to visually inspect which actors are central and which are not.

Another metric that would help to examine diffusion is adoption rate. In network structure, we can see how adoption rate constructs a network. The higher adoption rate, the faster a network is built. Adoption rate is usually measured by counting number of new adopters under some periods of time. It represents the number of adoption units who decide to positively accept an innovation in a given period. Adoption rate indicates how well the
innovation (in form of a software) is received by potential adopters. The higher the value of adoption rate, the better received that innovation is.

Adoption happens when a potential adopter perceives the benefit of using the innovation exceeds the cost of change (Hall 2004; Rogers 1995). In a system that exhibits network effect, a high adoption rate also implies decreasing uncertainty as more adopters use the innovation. Naturally, a high adoption rate implies a successful diffusion of innovation.

When a developer decides to adopt CakePHP, it implies that the benefits of using CakePHP exceed the cost of not adopting. It also decreases uncertainty for other potential adopters as more developers contributed to the project means there will be more support should the software go wrong. The adoption rate could as well be an indication of how successful CakePHP is, especially when compared to other similar projects.

As discussed in section 2.3.1.4, successful diffusion of innovation in a social system may also be attributed to opinion leaders. The visualization of network structure shall give insight on which actors gain central position as opinion leaders. We will try to identify contributors who fall under the category of opinion leaders. This study classifies opinion leaders according to criteria proposed in Rogers (1995). He found that opinion leaders are more exposed to all forms of external communication, are more cosmopolite, have somewhat higher social status and are more innovative. These selection criteria are then translated to the network graph data. The results are described in Table 2.

Table 2: Criteria of opinion leaders

<table>
<thead>
<tr>
<th>Criteria of opinion leaders</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposed to communication</td>
<td>Has in-degree centrality score above average</td>
</tr>
<tr>
<td>More cosmopolite</td>
<td>N/A</td>
</tr>
<tr>
<td>Higher social status</td>
<td>Is or was a core contributor</td>
</tr>
<tr>
<td>More innovative</td>
<td>Has an above average number of commits</td>
</tr>
</tbody>
</table>

3.3 Regression Model and Measurements

Analyzing the determinants of productivity is extensively performed in a framework of production function framework based on Solow (1957). The general specification of technical change in production function is:

\[ Q = F(K,L;t) \]  (1)

\(Q\) represents output, \(K\) represents capital input and \(L\) represents labor inputs. The time variable, \(t\), appears in the function to allow for technical change throughout a period of time.

As has been discussed in sub-section 2.4, we are about to test the relation of production input variables of an OSS community to its output. So in this regression model we simply include 3 independent variables, which are individual work intensity, network effect, and community’s work intensity. The time variable in this thesis is represented by monthly time
series data gathered between 2005-2014. Hence, a proper estimator that can handle time-series model should be employed to analyze the data. We will choose which estimator is appropriate to be used in this thesis.

According to the data at hand, this thesis uses a special type of variable as its dependent variable. The dependent variable, modified lines of code, is count data. To handle regression of count data, estimators such as poisson, negative binomial, zero-inflated negative binomial, and to some extent, OLS regression (with a log-transformed dependent variable) can be used to analyze count data (Long & Freese 2006). Other estimators such as logit, probit or binomial estimators are not particularly suitable since they require the dependent variable to have Boolean value (either 0 or 1). They are more suitable to estimate the tendency of how independent variables influence a choice or that has two choices.

Our dependent variable suffers from overdispersion, where the variance is much larger than the mean. In this situation, poisson regression is inappropriate since it assumes the mean to be more or less equal with the variance. To handle this situation, literature suggests to either employ negative binomial or zero-inflated negative binomial. As the dependent variable does not have many zeroes (only 1 out of 104 observations), it provides little justification to use ZINB model. A likelihood ratio test of alpha equals zero provides chi-square value of 6.8e+06 with one degree of freedom. This strongly suggests that alpha is non-zero and negative binomial estimator is more appropriate than poisson estimator.

Based on the considerations above, Negative Binomial regression is applied as a method to estimate the effects of individual work intensity, network effect and community’s work intensity on production output according to the specified model. The reason for choosing negative binomial regression was because of the special feature of our dependent variable.

With negative binomial model, the basic model is formulated as follows:

\[
\Pr(y_t = y_t | X_{1t}X_{2t}X_{3t}, \sigma_t) = \frac{e^{-\lambda_t} \times (\lambda_t)^{y_t}}{y_t!} \]

Where,

\[\lambda_t = \exp(\beta_1X_{1t} + \beta_2X_{2t} + \beta_3X_{3t}) \times \exp(\sigma_t)\]

Where \(y_t\) is production output per month, \(X_{1t}\) is individual work intensity, \(X_{2t}\) is network effect, and \(X_{3t}\) is community’s work intensity. Each of the beta, \(\beta_j\), are the coefficient to be determined, and \(\exp(\sigma_t)\) is assumed to have a gamma distribution with mean 1 and variance alpha, which can be estimated from the data. Alpha is the estimate of over-dispersion parameter that reflects unobserved heterogeneity, which corrects for the over-dispersion (Long & Freese 2006). If alpha equals zero, the model reduces to the simpler Poisson model. If alpha is significantly greater than zero, then the data are over-dispersed and better estimated by negative binomial model than Poisson model.
Since we are interested in measuring the expected value of the count variable, literature suggests interpreting the magnitude of a predictor using factor change (Long & Freese 2006). Interpretation using factor change refers to a measurement method called Incidence Rate Ratio (IRR) rather than using raw beta coefficient. This is motivated by the fact that our response variable is number of occurrence per month, which by definition is a rate. A rate is defined as the number of events that occur per time (or space). In the original interpretation using raw coefficients, the coefficients were interpreted as the difference between the log of expected counts. Since the difference between the log of expected count can mathematically be interpreted as ratio, then we could also interpret the regression coefficients as the log of the rate ratio. Finally, the rate at which events occur is called the incidence rate; thus we arrive at being able to interpret the coefficients in terms of incidence rate ratios from the interpretation above.

### 3.3.1 Dependent variable

The phenomenon under study is the productivity of OSS community. As discussed in sub-section 2.4.1, two most common measures for software productivity are either measuring the number of lines of code or function points. Considering the advance experience level, skills, rigor of measurement process and knowledge required to measure other productivity measurement methods, this thesis will measure the production output by counting number of modified lines of code. This method enables simple and straightforward calculation of production output. Moreover, it also provides consistent measurement across any programming language and could serve as an absolute measure of production output (Bok & Raman 2000). Therefore, the number of lines of code is used as an approximation for production output in this thesis (Ramírez & Nembhard 2004; Boehm n.d.).

The dependent variable $loc_t$ consists of the sum of additions and deletions made on the source code during month $t$. Figure 2 shows the distribution of lines of code.

![Figure 2: Distribution of lines of code. The dependent variable is count data.](image)
3.3.2 Independent variables

Based on the underlying reasoning explained on sub-section 2.4.2, the independent variables in this thesis consist of three things: individual’s work intensity, network effect and community’s work intensity. Considering that individual work intensity manifests as commits made by developers, I then approximate individual work intensity with number of commits. Hence, $X_{1t}$ is measured by the total number of commits made in month $t$.

Since network effect may only occur in a system with a lot of users, I approximate the existence of network effect using the number of contributors involved in writing software in a given month. Hence, variable $X_{2t}$ is measured by the number of contributors who are active in the community in month $t$.

The OSS community is a type of organization that produces, manages, processes, verifies and implements versions and releases. Those activities are a big part of what occupies developers on OSS projects. To support those activities, they employed VCS system. On the community level, one particular function of VCS system is to record releases from the community. So to capture the increasing work intensity around a deadline, this thesis approximates the community’s work intensity with number of releases recorded in VCS in a given month.

There are four general types of releases that indicated in the version number of a software according to a widely used practice using semantic versioning (Semver) from semver.org (Preston-Werner n.d.). Semver recognizes major version, minor version, patch version and development version. Major version is released when the developers make an incompatible API changes with prior version. Minor version is released when the developers add new functionality in a backward-compatible manner. Patch version is released when the developers make backward compatible bug fixes. All major, minor, and patch version is a stable and ready for production environment. A more unstable but includes experimental functionality is development version. This version is not ready for production environment.

This thesis does not account for different types of versions. We compile all versions in a variable called ‘releases’ to see its significance, effect and magnitude to the community’s production output. Hence, $X_{3t}$ is measured by the number of releases made by the community at month $t$.

The summary of production inputs and its correlating independent variables can be found in Table 3.

Table 3: Variables and description of productivity

<table>
<thead>
<tr>
<th>No</th>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Production output ($Y$-variable)</td>
<td>Number of modified lines of code (the sum of additions and deletions) in a given month</td>
</tr>
<tr>
<td>2</td>
<td>Work intensity of individuals</td>
<td>Number of commits made in a given month</td>
</tr>
</tbody>
</table>
### 3.3.3 Correlation matrix and descriptive statistics

The correlation matrix and descriptive statistics for all variables can be found in Table 4. We can see that our dependent variable, lines of code, is positively correlated with number of commits. While the number of contributors and releases are negatively correlated with lines of code.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Loc</th>
<th>Commits</th>
<th>Contributors</th>
<th>Releases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc</td>
<td>1.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commits</td>
<td>0.2182</td>
<td>1.0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contributors</td>
<td>-0.0212</td>
<td>0.4739</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>Releases</td>
<td>-0.0535</td>
<td>0.2701</td>
<td>0.5805</td>
<td>1.0000</td>
</tr>
<tr>
<td>Observations</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>Mean</td>
<td>31199.12</td>
<td>155.5385</td>
<td>11.21154</td>
<td>.7884615</td>
</tr>
<tr>
<td>Variance</td>
<td>8.01e+09</td>
<td>7684.27</td>
<td>55.64414</td>
<td>1.022778</td>
</tr>
<tr>
<td>S. Deviations</td>
<td>89473.3</td>
<td>87.65997</td>
<td>7.4595</td>
<td>1.011325</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>735191</td>
<td>368</td>
<td>31</td>
<td>5</td>
</tr>
</tbody>
</table>

Some high correlations from the data at hand are recorded among releases and contributors with 0.58 and also among contributors and commits with 0.47. These two largest correlation values are quite moderate and still at an acceptable level, hence we do not need to worry if there will be any variables dropped during regression.

All of the variables observed are fully complete without any missing values. There are in total 104 observations made, each representing one month timeframe. From this descriptive statistics we can also see that the variance of our dependent variable, $\text{loc}$, differs greatly compared to its mean. This means that our dependent variable clearly suffers from overdispersion.
4 Result

This chapter focuses on the productivity of CakePHP by presenting the regression result of individuals’ work intensity, network effect and community’s work intensity on the rate of production output. We start with the first part with an overview of CakePHP’s growth for the whole observed years. It examines growth pattern exhibited by the community and compares it with the common S-shaped diffusion pattern. Then we move to present regression findings for each of the predictors through every period. After that we move to the discussion of emergence and growth period in each following section. By the end of this chapter we reflect our regression results on the hypotheses presented earlier.

4.1 Overall growth pattern

As discussed in section 3.2, diffusion process may be visualized with a graph consisting of percentage or number of adopters plotted over a period of time. To see the diffusion pattern, we will look at the overall cumulative growth and active contributors in this subsection.

4.1.1 Cumulative growth

Figure 3 is CakePHP’s diffusion graph where the cumulative number of its contributors is plotted against time of adoption. A quick visual inspection on the graph shows that it resembles a familiar beginning of an S-curve.

![Cumulative contributors of CakePHP](image)

Figure 3: Cumulative contributors of CakePHP. Growth phase started from around August 2009

The adoption curve began with quite flat adoption line but at halfway of the inspected time period, its adoption line started to grow much steeper. This steep adoption rate indicates rapid growth as more contributors join the community. The steep adoption line does not seem to get flattened at the end of inspected time period, suggesting that the process
of diffusion is still going on. Based on this result, it is apparent that CakePHP has not entered maturity phase yet.

In addition to the differences in cumulative number of contributors, the two identifiable phases exhibit other significant compositional differences. We compare the number of new contributors (for adoption rate) and number of active contributors (for usage rate). We see that on growth phase both adoption rate and usage rate increases significantly compared to emergence phase. Figure 4 depicts the number of new contributors of CakePHP (marked with blue bars) and the number of active contributors (marked with black bars) between May 2005 and January 2014 per year.

![Figure 4: New contributors (blue) and active contributors (black) per year. Year 9 is not a full year cycle.](image)

The detailed data for the graph on Figure 4 above can be found in Table 5.

<table>
<thead>
<tr>
<th>Year</th>
<th>Active</th>
<th>Average monthly active</th>
<th>New</th>
<th>Avg monthly new</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1</td>
<td>4</td>
<td>1,50</td>
<td>4</td>
<td>0,33</td>
<td>4</td>
</tr>
<tr>
<td>Year 2</td>
<td>7</td>
<td>4,08</td>
<td>6</td>
<td>0,50</td>
<td>10</td>
</tr>
<tr>
<td>Year 3</td>
<td>15</td>
<td>6,50</td>
<td>8</td>
<td>0,67</td>
<td>18</td>
</tr>
<tr>
<td>Year 4</td>
<td>16</td>
<td>9,75</td>
<td>4</td>
<td>0,33</td>
<td>22</td>
</tr>
<tr>
<td>Year 5</td>
<td>43</td>
<td>10,75</td>
<td>32</td>
<td>2,67</td>
<td>54</td>
</tr>
<tr>
<td>Year 6</td>
<td>52</td>
<td>10,83</td>
<td>34</td>
<td>2,83</td>
<td>88</td>
</tr>
<tr>
<td>Year 7</td>
<td>104</td>
<td>21,08</td>
<td>74</td>
<td>6,17</td>
<td>162</td>
</tr>
<tr>
<td>Year 8</td>
<td>104</td>
<td>19,58</td>
<td>66</td>
<td>5,50</td>
<td>228</td>
</tr>
<tr>
<td>Year 9</td>
<td>79</td>
<td>19,63</td>
<td>42</td>
<td>5,25</td>
<td>270</td>
</tr>
</tbody>
</table>

From the introduction of CakePHP until August 2009, the project only managed to attract 4 to 8 new contributors per year. Until that time, there were only 23 contributors in total.
who are involved in the development of CakePHP. This low adoption rate exemplifies an emergence phase of diffusion, marked by low adoption rate, long innovation period, and mostly driven by innovators and early adopters (Rogers 1995).

CakePHP started leaving slow adoption rate and entered growing phase from around August 2009. Considering the late rapid adoption, CakePHP resembles the second diffusion pattern according to Rajagopalan et al. (2010), where the ‘takeoff’ seemed to be delayed after a period of time. In CakePHP’s case, it took four years to arrive at the takeoff point. This growth pattern exhibited by CakePHP can also be observed in other OSS projects, as documented in Rajagopalan et al. (2010).

Since entering growth phase from September 2009, the growth of contributors was increasing steadily with 33 people joined per year on average, for two consecutive years. It increases more than 8 times the amount of average contributors per month on Year 4. On the seventh year since first introduced, the growth peaked with twice the number of new contributors on the previous year, accounting for 74 developers (36%) of overall contributors. Its largest ever joining was in March 2012 and June 2013 where 28 new contributors joined. The period between August 2009 and June 2012 recorded the highest growth of new contributors in the community. The average was 6,17 new contributors joined per month.

After the number of new contributors peaked in 2012, the growth rate went generally slower. Then it fluctuates over the next coming months but on annual level the growth rate is going a bit lower at 60 contributors per year for the 8th and 9th year. However, this number is still higher than the 3rd and 4th year. By early 2014, the cumulative number of contributors increased more than twelve-fold compared to May 2009 (4th year) with 270 contributors in total. Figure 5 presents the graph of cumulative number of developers who joined CakePHP.

Although at the end of the observed period we find the adoption rate declines a bit, but this may not be an indication that CakePHP is entering maturity phase. In fact, it is still growing although not as fast. If the adoption rate continues to decline at that rate, then there is little doubt that CakePHP will enter maturity phase. However, should the community actively innovates then perhaps the growth will continue and will appear to be delayed as more people use CakePHP.

4.1.2 Active contributors growth

The number of monthly active developers informs similar situation. From the graph in Figure 4 we find the first three years underwent a slow growth with less than 10 active contributors each month. It is also apparent that the number of active contributors on 4th year is also low where the highest only records 16 contributors. It showed a decreasing trend towards July 2009 only to increase again afterwards.

On the growth phase, however, the number of active contributors per year jumps to 43 on the 5th year and surpassed 100 on Year 7 and 8. The average number of active developers per month increases two fold from Year 7 to Year 9. However, the trend is decreasing towards year 9, where around 20 developers were active compared to Year 7 where around 21 developers were active. The graph of monthly active developers and new developers per month are shown in Figure 6.
Based on the inspection we can be sure that emergence phase and growth phase has different activity rate. To get a detailed picture on what happens in the network during emergence and growth phase, we will explore the evolution of each phase in the following section.
Figure 5: Cumulative growth of contributors June 2005 - January 2014
4.2 Regression findings

The change triggered by growing labor input as we have seen in previous sub-section surely increases adoption rate and is expected to increase production rate to some extent as well. A legitimate question then would be to ask how much of that changes in the community is actually showing some productivity measurements. This sub-section seeks to measure how growing labor input enables work intensity and network effect to influence productivity. I examine their effect, sign, and magnitude on productivity using regression. This will validate our curiosity on the relation of diffusion and productivity.

To interpret how each predictor affect production output, the results of negative binomial regression for all periods, emergence phase, and growth phase are reported in Table 6. The first and second columns report regression result using IRR value for emergence phase (May 2005 – August 2009) and growth phase (September 2009 – January 2014), respectively. The last column reports the IRR value for predictors for the whole period. The regression result with raw regression coefficient can be found on Table 10 in the Appendix.

The overall regression result from the model proposed in section 3.3 has log-likelihood value of -1145.31. The model is statistically significant with probability of obtaining chi-square value at p<0.01. Also, we find likelihood ratio test that alpha equals zero - the likelihood ratio test comparing this model to a Poisson model has associated chi-squared value of 6.8e+06 with one degree of freedom. This strongly suggests that alpha is non-zero and hence enforces our decision of choosing the negative binomial as an appropriate model compared to Poisson model. As predicted, alpha is non-zero as its value is 1.373.

Table 6: Regression result with Incidence Rate Ratio (IRR) parameters for predictors, time-series negative binomial estimation.
Dependent variable: Number of modified lines of code per month from May 2005 – Jan 2014
The table reports IRR parameters with standard errors in parentheses.

For LR test of $\text{Alpha} = 0$, $\text{Prob} \geq \chi^2$ in parentheses.

Alpha: Over-dispersion parameter estimated with pooled models.

AIC = Akaike Information Criterion

*** p<0.01, ** p<0.05, * p<0.1

From the model for the overall period, it is found that all of the predictors are very significant at p<0.01. This is expected as we predict the work intensity of individuals, network effect, and work intensity of the community will affect productivity in a significant way.

Turning to the Incidence Rate Ratio (IRR) for each variable, it is found that the number of commits has positive effect on the rate of lines of code production. It is estimated that an increase of one commit in a given month would be expected to increase the rate of lines of code production by a factor of 1.01, while holding all other variables in the model constant. This is expected as we predict in H1 that the increase in individuals’ work intensity (approximated by number of commits) will increase productivity output (approximated by lines of code).

Interestingly, the number of contributors has unexpected negative effect on the rate of modified lines of code. It is estimated that an increase of one active contributor in a given month would be expected to decrease the rate of lines of code production by a factor of 0.928, while holding all other variables in the model constant.

Similar with the number of contributors, the number of releases also has negative effect on the rate of modified lines of code. It is estimated that if the community decides to make a single release in a given month, the rate of lines of code production would be expected to decrease by a factor of 0.705, while holding all other variables in the model constant.

The following section will comment on regression result for emergence phase, where the observations are limited to time period between May 2005 and August 2009.

4.3 Productivity on emergence phase

Turning to Table 6 again, the regression result for the first period has log-likelihood value of -547.42. The model is statistically significant with probability of obtaining chi-square value at p<0.05. Under this emergence phase, it is found that both commits and contributors are significant at p<0.01 and p<0.05, respectively. Meanwhile, ‘releases’ variable is found to be not significant at all during this period.

Turning to the beta coefficient for each significant variable, it is found that the number of commits has positive effect on lines of code. An increase of one commit in a given month would be expected to increase the rate of lines of code production by a factor of 1.008, while holding all other variables in the model constant. This is expected as individuals’ work intensity was predicted to have positive effect on the rate of lines of code production.
Meanwhile, number of contributors is again found to have negative effect on the rate of lines of code production. It is estimated that an increase of one active contributor in a given month during emergence period would be expected to a decrease the rate of lines of code production by a factor of 0.865, while holding all other variables in the model constant. It is interesting to note that both the magnitude and significance of network effect to the rate of production is at its lowest on this period.

Releases variable during this phase also have negative effect on the rate of lines of code production. This is similar with releases on the whole period. However, the coefficient is not significant, so one needs to be cautious about interpretation.

To sum up, the regression result for the first period means that on the emergence phase, contributors’ productivity is more defined by individuals’ work intensity and network effect rather than by community’s work intensity. On the next sub-section we continue with the regression result of productivity on the growth phase where the observations start from August 2009 until January 2014.

4.4 Productivity on growth phase

Turning to Table 6 again, the regression result under growth phase has log-likelihood value of -592.95. The model is statistically significant with probability of obtaining chi-square value at p<0.01. During this phase it is found that all predictors are very significant at p<0.01.

Turning to the beta coefficient for each significant variable, it is found that the number of commits again has positive effect on the rate of lines of code production. An increase of one commit during growth phase would be expected to increase the rate of lines of code production by a factor of 1.012, while holding all other variables in the model constant. When compared to the first period we can also see that productivity increases more rapidly for each addition of commit because on the first period we find lower rate of production.

Similar with previous phase, number of contributors is still showing negative effect on the rate of lines of code production. It is estimated that an increase of one active contributor during growth period would be expected to decrease the rate of lines of code production by a factor of 0.914, while holding all other variables in the model constant. It is interesting to note that the magnitude and significance of network effect on the rate of software production during this growth phase is higher when compared to emergence phase. Hence on average, contributors who join during this phase are a bit more productive than contributors who join on previous phase.

As for the community’s work intensity, it gains significance effect. It has negative effect on the rate of lines of code production. It is estimated that if the community decides to make an additional release in a given month, the rate of lines of code production would be expected to a decrease by a factor of 0.625, while holding all other variables in the model constant.
4.5 Results on hypotheses

On this sub-section we discuss the regression result presented in previous sub-sections. On all tested periods, individual work intensity (approximated by number of commits) is always found to be very significant and has consistent positive effect on productivity. It is also found that the individual work intensity is increasing in its magnitude on the growth phase when compared to emergence phase. This result means that in OSS community, individual work intensity is able to explain the increasing rate of production output. And since individual work intensity is positively linked to individuals’ intrinsic motivation and their expected immediate payoff, then we may conclude that intrinsic motivation and immediate payoff are indeed the sources of productivity at the individual level. This means that when developers find exciting and challenging problems to be overcome, they feel more stimulated to fix bugs and add new features, hence more code are contributed.

Second hypothesis deals with network effect as a variable that defines production output. It is found that network effect (approximated by number of contributors) has significant effect on all tested periods. This confirms our suspicions on Hypothesis 2. But interestingly, network effect is found to decrease the rate of software production.

The reason why network effect decreases production rate may be explained when we examine the characteristics of developers’ contributions. When we look at the distribution of commits made by each contributor in Figure 7, we find an enormous gap between the most productive and the least productive contributor. The distribution is heavily skewed towards the left side, which largely dominated by low-productive contributors who seldom make more than one commits nor write many lines of code. Since most developers only made one contribution, it is less plausible to argue that their contributions were motivated to pursue the benefits of having a large network. If their contributions were motivated by seeking delayed payoff such as peer recognition or better job prospect, it is more likely that they will contribute more regularly in order to increase their visibility on the eyes of other members of the community. But since most of developers only made one contribution during this phase, then they may simply be motivated only to solve their own problem instead of to pursue the benefits of network effect. This causes huge disparity in the composition of productive contributors by a large range. Hence when a new, average developer decides to join the community then she is more likely to drag down the average productivity level instead of increasing it.
Earlier in this thesis it was also suspected that network effect would have greater impact when the size of the network gets larger over a period of time. Time is of key importance since network effect would not be visible until enough number of users forms a ‘critical mass’ (Rogers 1995). According to the result from diffusion graph, more people are joining on the growth period compared to emergence period. This directly results in increasing size of the network. According to the regression result, we can see that network effect does indeed increase in both significance and magnitude (though still has negative effect) on growth period compared to emergence period. This result confirms our suspicion that network effect would be more apparent at a later stage of development.

Hypothesis 3 deals with the effect of community’s work intensity to the rate of software production. Release cycle is expected to have a significant and positive impact to the rate of software production since a scheduled release is suspected to increase the amount of modified lines of code written for that release. According to the regression result, we find that release has significant effect in overall period and on the growth phase, but not on the emergence phase. Therefore one needs to be cautious on interpreting the result on emergence phase. Moreover, the coefficient of the IRR also shows negative instead of positive effect. This may have been caused by two possible reasons. First, the type of releases is mostly dominated by patch version release (69%). The patch version release mainly deals with making backward-compatible bug fixes (Preston-Werner n.d.). So it is possible that prior to these releases, development activities mainly consist of refactoring codes and eliminating bugs. These two activities are somewhat less intense than writing new features so it may reduce the total number of modified lines of code prior to the release. Considering the significant difference release types may have on the rate of software production, it will be interesting to observe which type of release produces most code.
The second reason is possibly because for a period of time after a release, the software is believed to be stable enough so that no bugs are yet to be reported. Only after deployment in different environments the software may found to be unstable. The existence of a gap between an actual release and its subsequent reported bugs may delay software writing hence result in less number of modified lines of code. So it may also be useful to divide the analysis to prior to and after a version release. Table 7 presents the result of hypothesis testing we did by regressions.

Table 7: Hypothesis testing result

<table>
<thead>
<tr>
<th>Hypothesis testing</th>
<th>Emergence phase</th>
<th>Growth phase</th>
<th>All period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypothesis 1</td>
<td>Confirmed</td>
<td>Confirmed</td>
<td>Confirmed</td>
</tr>
<tr>
<td>Hypothesis 2</td>
<td>Confirmed</td>
<td>Confirmed</td>
<td>Confirmed</td>
</tr>
<tr>
<td>Hypothesis 3</td>
<td>Null hypothesis cannot be rejected</td>
<td>Significant, negative effect</td>
<td>Significant, negative effect</td>
</tr>
</tbody>
</table>
5 Analysis and Discussion

From earlier discussion we suspect that diffusion plays a large role to the productivity of OSS community. The growth of the software is an important issue since it may indicate the growing rate of innovation within the OSS community. As argued earlier, the growing rate of innovation shall result in the increase of productivity, which may be caused by network effect. Although the regression results show network effect decreases the rate of production output, we shall investigate the way it decreases production rate through visualizing its network structure. In this sense, we will explore how individuals are connected to each other in the community’s network as an indication of increasing productivity.

The first section identifies factors that determine the process of diffusion in the community of CakePHP. Specifically, it discusses the significance of key individuals and roles in the process of diffusion. On the second sub-section I examine the growth of network to explore how relationships between individuals evolve over time on every observed phases. Finally, at the end of this chapter we discuss the implication of CakePHP’s diffusion on productivity and provide some suggestions that may increase its productivity.

5.1 Determinants of diffusion

As discussed earlier, diffusion of innovation essentially deals with reducing uncertainty to encourage the adoption of innovation. An individual conducts information-seeking and information-processing activities to reduce uncertainties on the advantages and disadvantages of the innovation (Rogers 1995). This section explores how the role of leaders and social structure reduce uncertainties for new contributors to adopt CakePHP.

5.1.1 Role of leaders

Leadership is one determinant of diffusion that would reduce uncertainties in adopting new innovation (Lerner & Tirole 2002). The role of opinion leaders in the diffusion of an innovation is found to be crucial (Fogel 2006; Kazman & Chen 2009). We discuss the role of leaders especially opinion leaders and community leaders in the network of CakePHP to help it diffused among developers.

5.1.1.1 Opinion leaders

This study categorize the role of opinion leaders in an open source community according to criteria discussed in section 3.2. The criteria of opinion leaders in CakePHP community should satisfy all these criteria:

1. The score of inward degree centrality should be above average (average = 222.6)
2. Should be a core contributor
3. Has number of commits above average (average = 53.3)

When conditions above are applied to the data, only 15 people out of 270 contributors satisfy the conditions to be considered as opinion leaders. The names, in-degree centrality score, and number of commits are mentioned in Table 8.
Table 8: Opinion leaders in CakePHP

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Commits</th>
<th>In-degree centrality</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mark_story</td>
<td>5040</td>
<td>14930</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>2</td>
<td>PhpNut</td>
<td>1559</td>
<td>7746</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>3</td>
<td>Jose</td>
<td>1202</td>
<td>9501</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>4</td>
<td>nate</td>
<td>1054</td>
<td>2706</td>
<td>Past core contributor</td>
</tr>
<tr>
<td>5</td>
<td>gwoo</td>
<td>975</td>
<td>2785</td>
<td>Past core contributor</td>
</tr>
<tr>
<td>6</td>
<td>Juan Basso</td>
<td>455</td>
<td>3319</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>7</td>
<td>ADmad</td>
<td>358</td>
<td>790</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>8</td>
<td>AD7six</td>
<td>341</td>
<td>2967</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>9</td>
<td>Mariano</td>
<td>295</td>
<td>628</td>
<td>Past core contributor</td>
</tr>
<tr>
<td>10</td>
<td>euromark</td>
<td>271</td>
<td>838</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>11</td>
<td>Ceeram</td>
<td>270</td>
<td>1006</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>12</td>
<td>predominant</td>
<td>241</td>
<td>7010</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>13</td>
<td>Renan</td>
<td>195</td>
<td>973</td>
<td>Active core contributor</td>
</tr>
<tr>
<td>14</td>
<td>the_undefined</td>
<td>149</td>
<td>251</td>
<td>Past core contributor</td>
</tr>
<tr>
<td>15</td>
<td>DarkAngelBGE</td>
<td>119</td>
<td>677</td>
<td>Past core contributor</td>
</tr>
</tbody>
</table>

5.1.1.2 Community leaders

In the community of CakePHP, three contributors were known to have led its development namely Pies, PhpNut and Mark_story. Although Pies was very innovative, but has a very limited role in diffusion of CakePHP. The other two are opinion leaders. Characters of the three lead developers shall be discussed in the following paragraphs.

Pies is the original innovator of CakePHP as he was the one who initially written and released the code to public. Although he made initial effort to attract other innovators by releasing his code to public, it is apparent that he was less of an opinion leader. Three contributors joined during Pies’s leadership but two only made direct connection with him. He was rather focused on getting his creation to public than attracting new contributors or managing projects. That is the reason why Pies does not qualify to be included in Table 8 as an opinion leader. This case confirms what Rogers (1995) suggests that the most innovative member of a system often have deviant status of low credibility. Hence Pies is having limited role in diffusion of CakePHP.

After Pies left the community, PhpNut took over the position as the lead developer and continues the development for the next few years. In the first and second year PhpNut was also less of an opinion leader. This was due to the fact that the major priority of the community in 2005 and 2006 was to release a stable software instead of to acquire new contributors (Woodworth 2008). PhpNut did so by developing CakePHP solely by himself until he alone released the first stable major version.

Three new contributors joined after the first major release. PhpNut received valuable supports especially from Nate and Gwoo who were also very keen to contribute. They worked together on CakePHP not only by writing software codes, but more importantly they...
built CakePHP Bakery (a knowledge sharing system), CakeForge (platform to build upon CakePHP), Bin (a helper for IRC users), active in IRC, and more importantly, they built Cake Software Foundation. This effort was proven to be somewhat effective with positive response from the user community and four new opinion leaders were acquired.

Figure 8 shows network structure of opinion leaders under the lead of PhpNut. Twelve contributors were found to be opinion leaders of the community. The structure is more decentralized where most people are directly connected to each other. Lies in the center of the network are AD7six, DarkAngelBGE and Gwoo. Other opinion leaders including PhpNut are positioned at the edge of the network. PhpNut’s position at the edge of the network means that he made less connection with others therefore he was less connected. Instead of direct connection with PhpNut, other opinion leaders are connected with less technical, more managerial contributors such as AD7Six, DarkAngelBGE and Gwoo. Gwoo himself was appointed as project manager, thus it is natural to find him lies in the center of the network. PhpNut himself made most contributions in technical aspect of the project, which is reflected in his records of number of commits and lines of code. The role of PhpNut as the lead developer who concerns more about technical aspects earns him respect and trust from all his peers. Nevertheless, he decided to retire from the development of CakePHP in 2008. When PhpNut decided to stand down from the position as lead developer, many of core contributors who worked with him stopped to contribute as well. To continue the development, Mark_story took the responsibility to direct the next release as lead developer.

Mark_story made his first contribution on May 2008. His initial motivation to join the community was mainly motivated by altruism. This view is reflected in his personal blog when he was invited as a core developer “… it [open source] wouldn’t exist without countless hours of work. Contributing to open source will help keep it going, … Not only will giving back to open source give you that mushy good feeling, but you will also meet a lot of great like minded people who may end up connecting you with interesting paying work.” (Story 2008). It is apparent that Mark_story was mainly motivated by extrinsic motivation to give something back to the community that has given him opportunity to learn. Furthermore, he also acknowledges that getting involved in an OSS project will pull one into a circle of professionals who may somehow lead to commercial work.

As we discussed earlier, the diffusion of CakePHP took off around August 2009, a few months since led by Mark_story. Many new developers joined the community at a much
more rapid rate than it was. To manage the growing community, Mark_story received help from other new opinion leaders like AD7Six, Renan and Predominant. Not only that, they also facilitated the transition from past contributors who resigned from the community into a wave of new active contributors. Mark_story and friends managed to continue the reign of PhpNut in better ways.

Figure 9 shows network structure of opinion leaders under the lead of Mark_story from September 2008 until January 2014. Thirty contributors were found to be opinion leaders of the community during that time. The structure is somewhat centralized towards Mark_story, AD7six and Renan. On top of the network graph we can find a cluster of early contributors while bottom cluster represents new contributors. From the graph it is found that the three central contributors is perceived to act as communication agent between past contributors and new contributors.

5.1.2 Role of social structure

One determinant of diffusion that reduces uncertainties is the existence of social structure, as it would give potential adopters more predictability, structure and information about the innovation (Rogers 1995). We discuss the role of social structure on the following paragraphs.

Figure 10 depicts composition of new contributors divided by roles over the years. Black bars represent core contributors while blue bars represent ordinary contributors. During the first year, every single contributor became core developers regardless of the effort they put into the software. This number decreased slowly for the subsequent 3 years and plummeted at Year 5. More than half of new contributors on Year 5 only participate as ordinary contributor. The composition decreases even further with core developers are being dominated for the subsequent years until there is no new contributors joined core team by the middle of Year 9.
During emergence phase (Year 1-4), this high proportion of newly joining core contributors may indicate an effort from the community to reduce uncertainty. Almost everyone who contributed code to the main repository on that period obtained an important position as core developer.

One plausible reason for the easiness to join at an early stage would be that core contributors are of essential for development at an early stage rather than at later stage. The community needs more stable release thus they tried to acquire all the help they could obtain. It is naturally expected that having more core contributors will result in more stable release as more skill and expertise are put into the software in development. Stable release was the prime concern of the community at that time because stable software tends to be better accepted by most users. Thus, having more core developers in the community should increase the quality of the software. In accordance to increasing quality, trust will also increase and it means reduced uncertainty for potential users to adopt the innovation.

Another reason would be that the reward of joining at an early stage is higher than at a later stage. Joining the core team would be very beneficial for a contributor’s career, especially when the project is successful in the future (Lerner & Tirole 2002). The reward of being given higher status may reflect the community’s expectation to obtain new contributors’ commitments.

Although more rewarding, but the number of contributors who joined at an early stage is very small compared to later stage. This may due to high uncertainty involved in the project. That is why core contributors who joined at early stage are attributed to be more innovative and willing to take risks. Indeed, innovators and early adopters are usually regarded as more innovative and more venturesome (Rogers 1995).

Having some core developers also give clearer information on CakePHP’s organizational structure. Individuals who directly support the software will be visible to other potential adopters. Individuals who already had some reputation in other communities prior to joining CakePHP could increase the software’s credibility when they join the community. It was the case with PhpNut who were active in other online forums in relation
to PHP programming language. His reputation brought some new users, and new contributors to some extent.

5.2 Evolution of diffusion network

As discussed earlier, the growth of software may indicate the growing rate of innovation within OSS community. To show this, we will use network visualization by mapping the relations among contributors in the community. We will also inspect its evolution through the years. To do so, we will use cumulative network figures for each year so that the growth may be inspected annually. This section deals with the growth of diffusion using adoption rate to explain contributors’ acceptance of the tool.

Snapshot of cumulative network for each year is shown from Figure 12 until Figure 20. These pictures will be referred to when we discuss next sub-section. Green dots represent active core contributors, red dots represent past core contributors and gray dots represent ordinary contributor. Node size represents in-degree centrality, which means the bigger a node is, the more it connects to other nodes. Past core contributors are clustered to the upper-left part of the chart. It is interesting to note that active core contributors tend to be located centrally inside the network.

Figure 11: Dynamic visualization of CakePHP diffusion network can be accessed at Youtube.

The diffusion network evolved over time and actors who has central position vary for each period of time. One can view how actors join and leave the network in a period of one year, year by year from Figure 24 until Figure 32 in the Appendix. The dynamic visualization
of the network is shown in a movie like in Figure 11 and can be accessed on this Youtube link (or copy-paste this url in a browser: http://youtu.be/cbyDiTJtETA).

5.2.1 Diffusion on emergence phase

As discussed earlier, CakePHP underwent emergence phase during its first four long years of development. This long period of slow adoption rate is rather expected though, since it is common to find OSS communities started as a small individual project and only grows after found to be useful. This period of low adoption rate may be caused by several factors in relation with its communication channel, the readiness of CakePHP itself, time and social system. We shall examine the network structure annually to see how diffusion happens in a social system over a period of time.

The year 2005 until 2009 represents emergence period (Figure 12 until Figure 15). During this period, there were a very small but growing number of developers actively contributed to the main code repository. Three important stable versions of CakePHP were released during this period. The release was important to define the future of CakePHP.

![Figure 12: Cumulative network in Year 1, by May 2006](image)

![Figure 13: Cumulative network by May 2007](image)
During its first year in 2005-2006, only four contributors backed the development of CakePHP including the creator, Pies. The first three contributors joined in the first two months while the fourth joined later in the fourth month. No other contributors joined until May the next following year.

Looking at the background of the initial core team, they have no common background other than active in software development. Pies (the original creator) was an experienced web developer from Poland. He previously worked as a web designer at various technology companies in Poland for six years. PhpNut is a veteran OSS developer particularly in PHP from the US. He started active in software development since early 1990s. He contributed to well-known OS projects such as PHP-Nuke and OSCommerce. Prior to developing CakePHP, he served as core development team in PostNuke, a web content management system based on PHP. The other CakePHP’s core development team was Olle, from Sweden. He was mainly involved with writing documentation for Cake. Prior to CakePHP, Olle was a web designer started from the year 2000. As for Brego, he was a front-end web developer from Denmark. As all of the core developers have differences beyond geographical and educational background, we may imply that what unites them together was the same interest on web development technology. They already did some web development related jobs for years prior to contributing to CakePHP. Unfortunately, Pies did not involved for too long with CakePHP. The original creator left the community in July 2005 and the position of lead developer was taken over by PhpNut.
Although a number of developers joined in the beginning of the project, but their contribution is rather short-lived. Pies, Brego, and Olle did not contribute for long and the development of CakePHP reached its lowest rate in December 2005 where no one contributed any code. For the next following months CakePHP was developed solely by PhpNut without any contribution from the others. This situation occurred for 7 months until he alone released CakePHP 1.0 one year after the first introduction in May 2006 (Masters 2006). Following the major release, a minor version of 1.1.0 was released in less than a month time. CakePHP version 1.1.* has been a stable release for more than two years before a stable minor version of 1.2.0 came in December 2008.

After the release of 1.0 and 1.1 in a short period, three new developers namely Nate, Gwoo and Dho joined the community. The enrollment occurred in 2 months time after the release and no other developers joined for another 7 months. This suggests that the enrollment may be triggered by the major release. Gwoo was a very active opinion leader thus he got a position as project manager. Together with PhpNut he built the Bakery (knowledge sharing system), CakeForge (platform to build upon CakePHP), Bin (a helper for IRC users), active in IRC, and more importantly, they built Cake Software Foundation.

Another onboarding happened by March and April 2007 where Mariano joined among the other two. The average number of monthly active developers increases to 4.08 developers per month for the second year compared to only 1.5 developers per month in the first year. During this time PhpNut kept developing CakePHP along with Nate, Gwoo and Mariano. The three of them were very instrumental hence became core contributors as well.

By the third year (mid-2008), number of active developers grows very slowly and finally surpassed 10 active developers on May 2008. Between those times, the community continuously improves CakePHP version 1.1.* and there has been a steady increase on average number of active developer. However, the increase caused by this release did not yet show a ‘takeoff’ pattern suggested in Rajagopalan et al. (2010) nor satisfies a ‘critical mass’ criteria in Rogers (1995). Out of 18 developers who joined for the first three years, 15 of them (83%) became core contributor. Only three contributors proceed to contribute until 2014 while the rest are no longer active. The cumulative network by mid 2008 is shown in Figure 14. We can see that contributors from the first three years created a ‘clique’ at the far end of the network.

Year 4 (May 2008 - June 2009) was a critical year for the survival of CakePHP community. During this time there was a structural change occurred in the community where PhpNut as the lead developer was superseded by mark_story. Nate and Gwoo whose contributions are significant in helping PhpNut also resigned from the community in October 2009, a little bit later after Year 4. Nevertheless, newly joining developers on the following year quickly fill these empty positions so that the development of CakePHP continues. Here is how change happened in Year 4.

Figure 15 depicts the network of contributors in Year 4. Sixteen contributors were engaged in the development for the fourth year, 14 of which (87.5%) are core contributors. The change in organizational structure of CakePHP’s seems to affect the willingness for new developers to join the community negatively where only four new developers joined at that
time. Three out of them (Jperras, DavidPersson and Renan) became core contributors and gained quite central position in between all nodes in the network. Most of the core contributors from Year 3 were still in the network in Year 4 and became more connected than the previous year. The network structure on Year 4 was denser and the number of connections almost doubled compared to Year 3.

There were also some new versions of CakePHP released during Year 4. Minor version of 1.2.* was released in December 2008 after underwent a long 2 years of development. Its beta version was released in New Year of 2007, but slow to reach a stable release because of feature set issue. Like its predecessor, version 1.2.* also lived for quite a long time for around 1.5 years before got replaced with version 1.3.0 in April 2010.

Reflecting on why CakePHP did not exhibit a takeoff pattern during its first four years demands a look into its communication channel. Yet, establishing communication channels are the top priority of the community since the inception of CakePHP. Many communication channels were built since the first year to encourage interested developers to contribute. CakePHP registered their website since June 2005, just a month after it was first introduced. They used SVN as a VCS system to share code with others, created mailing lists group in Google, utilized a bug tracker (was located at http://trac.cakephp.org), created a knowledge sharing platform (the Bakery) and utilized real-time messaging channel using IRC. By the end of second year, the core contributors even founded Cake Software Foundation to ensure the software will have a strong support for years to come. All these effort to build various communication channels was done even before a stable version was released. Unfortunately, all these efforts to spread CakePHP did not result in a significant growth.

Another factor that would determine rate of diffusion is whether the innovation itself is beneficial for the potential adopters (Rogers 1995; Hall 2004). Potential adopters should perceive that the software’s benefits far outweigh its cost and uncertainty. An indicator that would lower the cost of adoption and uncertainty of software product is to release a stable version. CakePHP released the first stable version one year after first introduced. Although the product has been stable starting from its second year, it failed to see a rapid rate of adoption. It means that for the whole four years CakePHP software’s benefits and stability alone is not enough to attract new developers to contribute. Hence it failed to trigger the growth in adoption. This result suggests that the existence of a stable version cannot convince potential adopters to adopt CakePHP, so it is possible that the determining factor may be its network effect.

5.2.2 Diffusion on growth phase

Under this period a ‘takeoff’ pattern can be seen where new developers who joined the community increased sharply compared to previous period. Most of the change occurred from around the beginning of Year 5. We shall examine the annual network structure change from 2009-2014 to see how diffusion occurs during this period.
Figure 16 depicts the cumulative network of contributors in Year 5. There were in total 46 contributors engaged in the development for the fifth year and 19 of which (41.3%) were core contributors. This period marked the beginning of CakePHP’s growth phase where 32 new developers joined the community. Only 6 out of them (18.75%) became core contributors, and the rest were just ordinary contributors. The figure shows that newcomers in 2010 were more connected towards mark_story and predominant, and became less connected to PhpNut and the other past core contributors. In between these two main contributors, lies five core actors who connected them which are AD7six, Renan, Juan, predominant, and jose. The connections between past core contributors became less and less intense and two of them (joelmoss and phisy) did not continue to contribute this year. Only Nate, Gwoo and Jperras still maintain some intensive connections with other actors. In addition, new connections were made between ordinary contributors who joined that year.

The surge of growth pattern starting from August 2009 may be triggered by the structural change in the core development team. As has been argued in section 2, social structure determines the rate of which diffusion occurs. Core development team is a vital social structure since it consists of key individuals who have the power to drive the direction of the project. If the surge is indeed attributed to the change of social structure, then that indicates how opinion leaders and change agents play crucial role in diffusion.

Another possible determinant of this surge may be the release of another stable version (CakePHP version 1.2), which occurred early in January 2009. But this argument is weak for two reasons. First, the gap between this release and the surge was too long. The release occurred 8 months prior to the surge. Previous release of a stable version (version 1.0 and 1.1) only managed to attract new developers for the first two months after its release date then followed by 7 months of no new developers. Second, the release did not cause a surge in
the number of people joining. It only adds in total 3 new developers during that first 2 months. It is then arguable that the change was more likely to be triggered by structural change instead of stable releases.

Figure 17 depicts the cumulative network of contributors in Year 6. There were 63 developers engaged in the development for the sixth year and only 15 of which (23.8%) were core contributors. The number of new developers are increasing compared to the previous year but not by a large margin. One can witness a shift of central actors who writes code, where new contributors were replacing old ones. Some past core contributors (Nate, Gwoo and Mariano) were still in the network during this period but there were no connections between any of them. They were connected only to active core contributors but not to ordinary contributors. This indicates that they were no longer active in the community therefore their codes were being replaced with new ones.

Out of 34 new contributors participated that year only one (2.9%) was appointed to become core developer. During this period, the ones who hold distinguishable importance as central actors were Jose, Mark_story and AD7six. One of the core contributors was not connected to anyone during this year and it is an indication that his files were not yet modified by anyone.

During this period there were on average 20 developers actively contributed to the main code repository, almost twice from the previous period. The release cycle went faster with one major release in October 2011 followed with minor release every 5-6 months. It seems that the number of new joined developer during this period is decreasing with the highest amount were recorded during year 7 with 6.17 developers joined per month.
Figure 18 depicts the cumulative network of contributors in Year 7. There were 114 developers engaged in the development on the seventh year where only 15 of which (13.2%) were core contributors. In this period there were only two early core developers (PhpNut and Mariano) who contributed in the early days of CakePHP were involved in the development. Two past core developers (indicated by red dots, both named Daniel) were actually new contributors who were appointed as core developers but then stop contributing after only a short period of time thus they are classified as past core contributors. No other new contributors this year were appointed as core contributors. The network structure during this year was less centralized compared to previous year. Central actors who facilitate development process under this period were mark_story, Juan, Jose, and Ceeram.
Figure 19: Cumulative network by May 2013

Figure 19 depicts the cumulative network of contributors in Year 8. There were 110 developers engaged in the development for the eighth year and 12 of which (10.9%) were core contributors. In this period, none of the early core contributors took part in the development. The network of contributors consists fully of new contributors and being dominantly led by mark_story. Other actors who were helping mark were Jose, Ceeeram, ADmad and euromark.
Figure 20 depicts the cumulative network of contributors in Year 9. There were 83 developers engaged in the development for the ninth year and 12 of which (14.5%) were core contributors. During this year the network looks less populated because it only accounts for 8 months until January 2014 instead of 1 year. Nevertheless, the network during this year does not change much from the previous year and still dominated by mark_story although the role of other core contributors like predominant, Juan, Renan, Jose, AD7six and Ceeram is decreasing in terms of centrality. It is interesting to note that most of the core contributors who remain involved in the network until Year 9 were the ones who joined during the transition in Year 4 and Year 5.

5.2.3 Current state of diffusion

After going through almost 9 years of development, the current network structure of CakePHP by mid-January 2014 is depicted on Figure 21.
From the evolution of graph we learn that network of contributors of CakePHP is a dynamic network that exhibits a varying rate of adoption. Initially we see a small number of developers started the project and continues as a small team for a bit more than four years. At one point, we observe a rising growth in the network in 2009 where new contributors are adopting at a faster rate. This was when the network left emergence phase and entered growth phase.

The growth from emergence to growth phase shows some remarkable changes in the structure of developers. Newly active core contributors such as Mark story, predominant, AD7six, Renan and Jose lead the network during the growth phase. Any new contributors who joined on the growth phase are engaged with active core contributors who succeeded past core contributors. New versions and development is done with the lead of successive core contributors, and code from past developers is slowly replaced by new ones. We also observe that past core contributors who mostly joined during first four years (marked by red dots) created a ‘clique’ at the far end of the network. This separation between past and new contributors suggests that the increase in adoption was triggered by the organizational change occurred in 2009. Thus we suspect that the cause of more productivity is the newly appointed core contributors, since they bring new contributors to adopt CakePHP and in turn would enhance network effect. The role these distinguishable individuals and their roles are discussed in the following section.
5.3 Implications of diffusion to productivity

From the discussion in the previous section, both network and work intensity in each of the identified phases in the adoption graph was found to exhibit different characteristics. On the emergence phase we observe a low adoption rate and small network size. The community’s work intensity during this phase also did not occur very intensively. Only three stable versions were released during emergence period. Individuals’ work intensity is also found to be low, especially during the first year when PhpNut was the only one who developed the software for several months.

On the growth phase we observe a high adoption rate and hence bigger network size. More people were connected towards each other and the network structure was quite centralized towards core contributors. The community’s work intensity during this phase also increases in quantity where one major and five minor versions of CakePHP were released. Even more so with individuals’ work intensity where it was found that more contributors are more active compared to the prior phase.

Coming back to our discussions on the disparity in contributors’ contributions, we found that the disparity is very visible at the period where network effect is very significant, that is, at growth phase. The cumulative network from 2010 to 2014 (Figure 16 to Figure 20) shows that the actors who drive the productivity during growth phase are the ones who are located in the middle of the network. These twelve key contributors and organizers of the community (namely Mark_story, Predominant, Juan, Jose L, Jose D, Ceeram, Renan, AD7six, ADmad, Euromark, Rachman and to a lesser extent, PhpNut) are the ones who contribute to 87% of all the commits made during that period. As a comparison, the other 245 contributors only contribute to 13% of lines of code for that period. Hence, high-performing contributors can easily trump the low-performing ones in terms of productivity.

Increasing OSS productivity

To stimulate productivity that is based on individuals’ work intensity, I reflect to the argument in Lerner & Tirole (2002). Initial leaders must provide critical mass of code so developer community can react to it. Enough work must be done to show the project has merit and beneficial. They also need to leave challenging tasks and actively encourage contributions to attract new developers. By doing so, potential adopters developers may identify exciting problems to solve in the project hence decide to join the community.

Although network effect was suspected to increase production output, it only happens on the aggregate level, not on average. An addition of new contributor increases the total production output, but the average production output per person decreases considering the disparity of productivity between contributors. To reduce the effect of this situation, the project needs to attract committed individuals who are willing to contribute continuously. Attracting commitments may be done by convincing new individuals about the delayed payoff that builds up after a period of contributions. Only outstanding individual with above average performance may reap the reward of peer recognition or better job opportunity.
Unfortunately in this thesis we are unable to investigate the productivity based on heterogenic individuals’ skill level because of data limitation.

The last variable we consider to investigate is community’s work intensity. According to our regression result, the community’s work intensity (approximated by release) affects production output in a significant way. Intensive release is beneficial to spot bugs and attract more developers to use it (Raymond 1999), but the activity of fixing bugs and refactoring code itself does not necessarily increase the rate of software production in terms of lines of code.
6 Conclusions

The purpose of this study is to establish an understanding on the productivity of OSS developer community through diffusion of innovation. The strength of this thesis is to deliver a story line to the life cycle of a community through sequences of diffusion network presented in the form of network visualization graphs. It is important to understand whether the community’s diffusion pattern follows the familiar S-curve. The network visualization helps to explain relationships between contributors and its dynamics in a social system of diffusion.

Our first research question asks how does diffusion affect the productivity of OSS developer and whether the OSS community’s diffusion pattern follows the familiar S-shaped curve. Based on the discussion on its diffusion pattern, we found that the adoption of CakePHP does indeed follow the S-shaped diffusion curve commonly exhibited in traditional innovation life cycle, although yet to finish a full life cycle. Its S-shape is characterized by slow adoption during emergence phase and then followed by rapid adoption during growth phase. However, the growth phase does not seemed to end yet and the maturity phase is still not visible in the graph.

Both diffusion phases shows different network characteristics. It is apparent that in the growth phase there are increases in adoption rate, network size, number of connections, total production output, and work intensity when compared to emergence phase. Some contributors, especially opinion leaders and community leaders played a crucial role to drive adoption. During transition period from emergence phase to growth phase, we witness how contributors who joined during early years are replaced by the new ones after 3 to 4 years. It is interesting to note that the composition of core contributors decreases as a sign of decreasing uncertainty and increasing community members.

It is also observed that network structure of CakePHP plays an important role in diffusion and its productivity. In this case the governing body consists of people who are highly skilled and highly productive. As can be seen from diffusion network that most contributors are connected to leaders of the community, making the leaders’ positions very central to connect new developers. In this sense, the appearance of key individuals is an important aspect that determines CakePHP’s productivity growth. The leaders are able to attract and connect new comers with the whole community. Two community leaders are found to be very instrumental in shaping the growth of CakePHP’s adoption, namely PhpNut and Mark_story. They managed to not only deal with the technical aspect bust also the managerial aspect of CakePHP. The community also promoted a number of active contributors as core developers in its governance structure as an effort to maintain stability and reduce uncertainty.

Our second research question deals with how individual’s work intensity, network effect and community’s work intensity affect productivity. The regression results show that individual work intensity (approximated by number of commits) is found to be very significant and has consistent positive effect on the rate of software production at all phases. In addition to that, network effect is also found to have significant but negative effect to the
rate of software production at every period. But as have been discussed before, this negative effect is very likely to be caused by the heterogeneity of contributors’ productivity.

In CakePHP case, it is found that network effect does exist but its existence does not consequently result in the increase on the rate of software production, as most developers tend to contribute only a few times. This result suggests that network effect is beneficial to attract new contributors but does not guarantee long-term commitment.

To our surprise, community’s work intensity (approximated by number of releases) is found to have significant but negative effect on the rate of software production on overall period and growth phase but not on emergence phase. It is suspected that this effect may either be caused by the heterogeneity in the composition of types of releases or be caused by relatively stable software following a release. To account for those issues, future research agenda shall include release types and time-lag parameter on its predictors.

Directions for future research

This thesis has discussed the issue of productivity by capturing the OSS community as a system of production. The regression model used in this thesis shows a good fitness although still shows signs of overdispersion. One way to overcome the issue of overdispersion will be to specify a more granular level of analysis with more variables. Perhaps the result will get better if we could investigate productivity on individual level in a monthly basis with panel data. With monthly individual production rate as dependent variable, it will be possible to add new variables on individual level such as skill, education, work, age, etc and on community level such as types of releases produced in a given month. Perhaps the increase in number of variables and number of observations will allow for better accuracy and better robustness.
References


Github, 2013. No Title. Available at: https://github.com/cakephp/cakephp.

Gitref, Git Reference. Available at: http://gitref.org/basic/ [Accessed April 21, 2014].


Oikarinen, J. & Reed, D., 1993. Internet Relay Chat Protocol,


## Appendix

Table 9: Annual growth of commits

<table>
<thead>
<tr>
<th>Year</th>
<th>Commits</th>
<th>New</th>
<th>Avg monthly new</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1</td>
<td>345</td>
<td>345</td>
<td>28,75</td>
<td>345</td>
</tr>
<tr>
<td>Year 2</td>
<td>1539</td>
<td>128,25</td>
<td>1884</td>
<td></td>
</tr>
<tr>
<td>Year 3</td>
<td>1718</td>
<td>143,17</td>
<td>3602</td>
<td></td>
</tr>
<tr>
<td>Year 4</td>
<td>1450</td>
<td>120,83</td>
<td>5052</td>
<td></td>
</tr>
<tr>
<td>Year 5</td>
<td>2557</td>
<td>213,08</td>
<td>7609</td>
<td></td>
</tr>
<tr>
<td>Year 6</td>
<td>3042</td>
<td>253,50</td>
<td>10651</td>
<td></td>
</tr>
<tr>
<td>Year 7</td>
<td>2910</td>
<td>242,50</td>
<td>13561</td>
<td></td>
</tr>
<tr>
<td>Year 8</td>
<td>1605</td>
<td>133,75</td>
<td>15166</td>
<td></td>
</tr>
<tr>
<td>Year 9</td>
<td>1010</td>
<td>126,25</td>
<td>16176</td>
<td></td>
</tr>
</tbody>
</table>

Table 10: Regression result with raw coefficient parameters

<table>
<thead>
<tr>
<th>Lines of code (loc)</th>
<th>Emergence Phase</th>
<th>Growth Phase</th>
<th>All period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits</td>
<td>.0085975***</td>
<td>.0118932***</td>
<td>.0099154***</td>
</tr>
<tr>
<td></td>
<td>(.0028575)</td>
<td>(.0025809)</td>
<td>(.0016235)</td>
</tr>
<tr>
<td>Contributors</td>
<td>-.1455802**</td>
<td>-.0897171***</td>
<td>-.0746094***</td>
</tr>
<tr>
<td></td>
<td>(.0705403)</td>
<td>(.0344954)</td>
<td>(.0229911)</td>
</tr>
<tr>
<td>Releases</td>
<td>-.1050964</td>
<td>-.4705748***</td>
<td>-.3487647***</td>
</tr>
<tr>
<td></td>
<td>(.3261286)</td>
<td>(.1487712)</td>
<td>(.1226345)</td>
</tr>
</tbody>
</table>

Dependent variable: Number of modified lines of code per month from May 2005 – Jan 2014.
The table reports coefficient parameters with standard errors in parentheses.
Figure 22: Cumulative commit growth June 2005 - January 2014
Figure 24: Network in Year 1, mid-May 2005-2006

Figure 25: Network in Year 2, mid-May 2006-2007

Figure 26: Network in Year 3, mid-May 2007-2008
Figure 27: Network in Year 4, mid-May 2008-2009

Figure 28: Network in Year 5, mid-May 2009-2010
Figure 29: Network in Year 6f, mid-May 2010-2011
Figure 30: Network in Year 7, mid-May 2011-2012
Figure 31: Network in Year 8, mid-May 2012-2013
Figure 32: Network in Year 9, mid-May 2013 - mid-January 2014