Two Stage DEA 를 통한

오픈소스 소프트웨어 프로젝트의 효율성 분석

- 깃(Git)에 의한 커밋 생산 과정을 중심으로 -

2017 학년도 1 학기 경영과학

김창희 교수님

경영학과 2016-14877

송재윤

목차

1. 연구 동기 및 필요성

오픈소스 소프트웨어(Open Source Software, 이하 OSS)란 단지 무료로 배포된 소프트웨어를 말하는 것이 아니다. 소프트웨어 자체의 사용 뿐만 아니라, 소프트웨어를 구성하는 소스 코드에 대한 접근권과 더불어, 누구나 코드를 수정하고 파생 저작물을 작성할 수 있도록 자유로운 재배포를 허용하는 개념이다(Androutsellis-Theotokis et al., 2010). 운영체제 'Linux'나 데이터베이스 관리 시스템 'MySQL', 웹 브라우저 'Chrome' 등이 대표적인 OSS 의 성공 사례이다. 이 밖에도 수많은 기업과 정부 기관에서 OSS를 이용하고 있으며, 우리나라 OSS 시장 역시 매년 약 15.2%의 성장률을 보이며 꾸준히 확대되고 있다.[1]

전통적 시각에서 이러한 OSS 프로젝트의 성공은 상당히 파격적인 현상이다. 불특정 다수가 경제적 이해관계에 기대지 않고 자발적인 동기에 의해 온라인으로 소통하며 협업하는 구조는 기존의 경제학적 관점으로 설명할 수 없다. Raymond(1999)는 "보는 눈이 많아지면, 모든 버그가 드러난다."[2] 는 리누스 법칙(Linus' Law)을 제시하며 OSS 성공의 핵심으로 '다수의 협업'을 꼽았다. 즉 대중이 직접 소프트웨어를 테스트하고 버그를 수정하거나 새로운 기능을 추가함으로써 폐쇄적으로 개발되는 상업용 소프트웨어를 능가하는 결과물이 생산된다는 것이다.

그러나 이러한 자율적, 개방적 환경 역시 한계를 가진다. 다수의 참여는 때로 집단의 효율성을 상대적으로 떨어뜨리기 때문이다(Steiner, 1972). 집단의 규모가 커질수록 비효율적 의사소통이 발생할 가능성이 늘어나고, 프로젝트의 원활한 관리가 어려워진다. 예컨대 주요 기능의 추가와 같은 높은 수준의 의사결정에 있어서는 다수의 참여가 오히려 독이 된다. 실제로 많은 OSS 프로젝트가 비효율로 인해 성숙하지 못한 채 실패하고 만다. Schweik & English (2012)에 따르면, 전체 OSS 프로젝트 중 성공적인 프로젝트는 단 17% 에 불과한 것으로 나타났다.

---

1 정보산업진흥원 통계, 2016

2 Given enough eyeballs, all bugs are shallow.

그렇다면 OSS 프로젝트가 효율적이기 위해서는 무엇이 필요한가? 단순히 다수의 적극적인 참여가 효율적인 OSS 를 보장하는 것은 아니다. 효율적인 OSS 프로젝트를 위해서는 다수의 참여를 원활히 관리할 메커니즘이 필요하다. 즉 적절한 권위를 부여 받은 소수 관리자의 다수에 대한 규제가 일정 수준 동반될 수밖에 없는 것이다(원인호, 2014). 본 연구에서 다루는 '깃(Git)'이 바로 이러한 자율적 협력을 위한 규제 장치의 일례이다.

따라서 OSS 프로젝트의 효율성에는 이러한 내부적 규제 프로세스의 반영이 필수불가결하다. 단순히 최초 투입과 최종 산출물의 값으로는 완전히 새로운 구조에 기반한 OSS 의 효율성을 나타낼 수 없다. 이에 본 연구에서는 기존의 관점을 버리고 새로운 시각을 취하면서, 자율적 협력을 위해 동반되는 일련의 과정을 반영함으로써 OSS 프로젝트의 효율성을 재정의하고자 한다. 이를 위해 Data Envelopment Analysis(DEA)를 활용하되, 내부적 프로세스를 반영할 수 있는 Two Stage Model 을 설정하였다. 또한 새롭게 정의된 효율성을 기준으로 Kruskal-Wallis 검정을 시행해 다수의 참여가 OSS 프로젝트에 어떠한 영향을 미치는지 알아보도록 하겠다.

2. 선행 연구

수년간 많은 학자들이 OSS 프로젝트의 효율성에 관심을 가지고 나름의 해석을 제시해왔다. 이중 본 연구가 주로 참고한 선행 연구는 Ghapanchi 와 Aurum 이 연구하고 2012 년에 발간한 SCI 급 경영 학술지 Electronic Markets 에 게재된 'Competency rallying in electronic markets: implications for open source project success'이다.

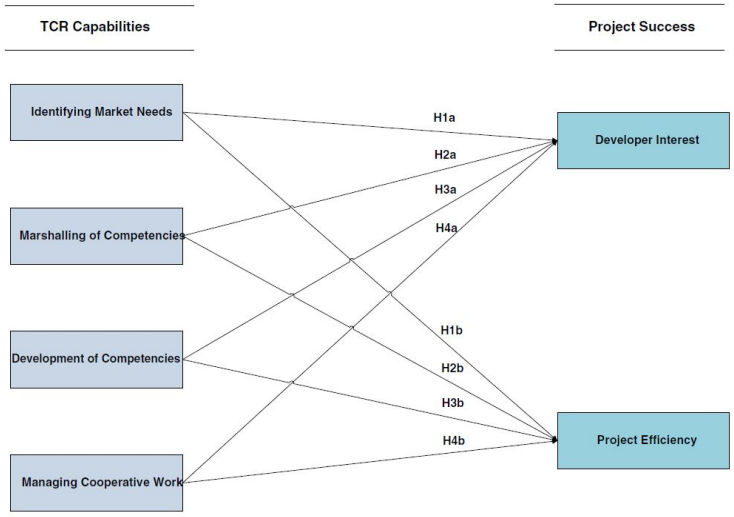

그림 1. Research Model of Ghapanchi & Aurum (2012a)

이 연구는 Partial Least Square(PLS)기법을 통해 Theory of Competency Rallying(TCR)의 네 가지 역량이 각각 OSS 프로젝트 성과에 대하여 가지는 영향을 조사하였다. OSS 의 맥락 속에서 TCR 은 새로 등장한 고객의 요구에 대응할 경쟁력을 갖춘 개별 개발자를 한 곳으로 모으는 과정을 다룬다. TCR 에서 제시한 성공적인 프로젝트의 네 가지 역량 시장의 요구 확인(Identification of Market Needs), 경쟁력 식별(Marshalling of Competencies), 경쟁력 개발(Development of Competencies), 그리고 협력적 업무 관리(Managing Cooperative Work)가 연구 모형의 독립 변수이며, 종속 변수에 해당하는 OSS 프로젝트 성과는 개발자의 관심도(Developer Interest)와 프로젝트 효율성(Project Efficiency)로 구성된다. 이때 프로젝트 효율성을 DEA 모델을 통해 분석했는데, 개발자의 수와 프로젝트 존속 기간을 투입으로, 출시한

파일의 수를 산출로 상정하였다. 또한 분석의 대상은 OSS 개발 플랫폼 '소스포지(sourceforge.net)'의 프로젝트이며, 동질성을 확보하기 위해 그 중 개발자용 프로그래밍 도구를 제공하는 '소프트웨어 개발' 카테고리의 프로젝트 607 개를 선정하였다. 이렇게 구해진 효율성을 바탕으로 PLS 분석을 실시한 결과, 시장의 요구 확인을 제외한 모든 TCR 역량이 프로젝트 효율성에 긍정적인 영향을 주는 것으로 나타났다.

이 밖에도 Ghapanchi & Aurum(2012b), Wray & Mathieu(2008), Koch(2009) 등이 DEA 를 활용하여 OSS 프로젝트의 효율성을 도출하였다. 아래 [표 1]은 본 연구가 참고한 선행 연구에서 사용한 분석 모델을 정리한 것이다.

연구	모형	Input		Output	
		Labor	Time	Quantitative	Qualitative
Ghapanchi & Aurum (2012a,b)	산출지향 BCC	개발자의 수	프로젝트 존속 기간	출시한 파일의 수	X
Wray & Mathieu (2008)	투입지향 BCC	개발자의 수 버그 신고자의 수	X	유용한 파일 크기	다운로드 횟수 프로젝트 순위
Koch (2009)	산출지향 BCC	개발자의 수	프로젝트 존속 기간	파일 크기 소스 코드의 줄 수	X

표 1. 선행 연구의 연구 모델

상업용 소프트웨어의 효율성의 경우 일반적으로 노동과 시간 두 가지의 범주로 투입을 구분한다. OSS 프로젝트 역시 두 가지 범주는 유지하되, OSS 의 개방적, 자율적 특성을 반영하여 약간의 조정이 이루어졌다. 우선 OSS 프로젝트에 대한 노동은 자발적으로 이루어지기 때문에 금전적 측정이 불가하다. 따라서 선행 연구에서는 투입으로 금전적 비용 대신 OSS 코드를 작성한 개발자의 수를 사용하였다. 그런데 넓은 의미에서 OSS 프로젝트에 투입된 인적 자원은 직접적으로 코드 작성에 참여한 개발자뿐만 아니라 버그를 신고하거나 새로운 기능에 대해 의견을 제시한 대중까지 포함한다고 볼 수 있다. 이에 Wray & Mathieu(2008)는 개발자의 수에 더하여 버그 제출자의 수를 노동 변수로 채택한 바 있다. 또한 선행 연구에서는 짧은 기간 동안 신속한 성장을 이룬 소프트웨어를 효율적인 것으로 보아, 시간을 나타내는 변수로는 프로젝트

존속 기간을 또 하나의 투입 변수로 선정하였다.

한편 산출의 경우 양적, 질적 지표로 구분해볼 수 있다. 양적 지표로는 주로 파일 크기, 소스 코드 라인 수(Koch, 2009), 소스 코드 속 함수의 개수(반승현 외, 2014), 출시한 파일의 수(Ghapnchi & Aurum, 2012a) 등이 있다. Wrau & Mathieu(2008)는 유용성을 고려하고자 파일 크기를 다운로드 횟수로 나눈 값을 사용하기도 하였다. 또한 질적 지표로는 다운로드 횟수와 프로젝트 순위가 사용되었다.

위와 같은 선행 연구들은 모두 상업용 소프트웨어와 명확히 구분되는 OSS 프로젝트의 효율성을 새로이 정의하는 데 기여했지만, 두 가지 부분에 있어 한계를 가진다.

첫번째 한계는 분석의 대상을 소스포지 프로젝트로 삼아, 최신 OSS 프로젝트에 적용하기에 간극이 존재한다는 점이다. 현재 소스포지는 단지 소프트웨어를 다운로드하기 위한 용도의 웹사이트로 변질되어 더 이상 OSS 플랫폼을 대표하지 못한다. 이는 2007 년 등장한 이래 급속도로 성장하여 최근 세계 최대의 OSS 프로젝트 플랫폼으로 등극한 깃허브(Github.com)의 영향이 크다. 깃허브의 OSS 시장 장악으로 인해 또 다른 OSS 커뮤니티였던 구글코드(Google Code)는 2015 년도를 마지막으로 서비스를 종료하기도 하였다. 본 연구는 OSS 프로젝트의 효율성을 측정함에 있어 깃허브를 기준으로 하여 보다 시의 적절한 결과를 도출하고자 한다.

두번째 한계는 OSS 프로젝트 효율성에 대한 내부 처리과정을 고려하지 않았다는 점이다. 특히 깃허브는 깃(Git)이라는 버전 관리 툴을 통해 훨씬 복합적인 개발 프로세스를 제공한다. 깃허브에 기반한 OSS 프로젝트는 참여자에 의해 작성한 코드가 커밋(Commit)을 생성하고 원본 코드와 병합(Merge)될 때까지 깃에 의한 일련의 심사 과정을 거친다. 전통적인 DEA 모델은 이러한 복합적인 과정을 내포하고 있는 OSS 프로젝트의 효율성을 정확히 담아내지 못한다. 투입과 산출 사이의 과정은 일종의 블랙박스(black box)으로 간주해버리기 때문이다(Tone and Tsutsui, 2009). 따라서 프로젝트 참여자들의 코드 작성이 중간 생산물을 거쳐 최종 산출물까지 전달되는 OSS 프로젝트의 내부 처리 과정을 고려하기 위해 확장된 DEA 모형을 적용할 필요가 있다. 이에 본 연구에서는 Two Stage DEA 기법을 분석 모형으로 사용하여, 전통적인 DEA 로는 찾아내기 어려운 내적 비효율성을 포착하고자 한다.

3. 연구 대상

3.1. 깃과 깃허브

본 연구는 OSS 개발을 위한 플랫폼인 '깃허브(Github.com)' 상에서 진행되는 프로젝트를 대상으로 한다. 깃허브는 2007 년 만들어진 이래 기하급수적으로 성장하여 소스포지와 구글코드 등 다양한 서비스를 제치고 세계 최대 규모의 OSS 프로젝트 플랫폼으로 자리잡았다. 이는 깃허브가 협업을 지원하는 다양한 기능을 무료로 제공하기 때문이기도 하지만, 무엇보다도 깃(Git)[3]이라는 버전 관리 시스템이 핵심이다. 버전관리 시스템이란, 여러 개발자가 동일한 소스 코드를 동시에 수정해도 문제가 발생하지 않도록 소스 코드의 모든 변화 이력을 관리하는 체계를 말한다. 즉 다수의 참여를 효율적으로 관리하기 위한 일종의 자율적 규제 장치인 셈이다. 깃에서는 소스 코드의 변화가 '커밋(Commit)'이라는 단위로 저장되며, 변화를 바로 반영하지 않고 일련의 메커니즘을 통해 단계적 심사를 통과한 코드만을 받아들일 수 있다. 깃허브의 OSS 프로젝트들은 모두 기본적으로 깃을 이용하여 작동한다.

깃허브에서는 깃을 통해 일반 개발자의 입장에서 OSS 프로젝트에 기여하는 과정을 6 단계로 나누어 설명하고 있다.[4]

(1) 가장 먼저 해야할 일은 기여하고자 하는 프로젝트에 브랜치(Branch)를 생성하는 것이다. 실제 배포될 OSS 에 해당하는 코드가 있는 곳을 마스터(Master)라고 하는데, 모든 수정이 마스터에서 한꺼번에 일어나는 혼란을 막기 위해 브랜치가 사용된다. 하지만 매번 브랜치를 만들 필요는 없다. 예컨대 댓글과 관련된 기능을 제안하고 싶은데 comment 라는 브랜치가 이미 존재한다면 해당 브랜치에서 작업하면 된다. 또한

---

[3] 재밌는 건, 깃 역시 또 하나의 오픈소스 시스템이라는 사실이다. 오픈소스의 대가 리누스 토발즈(Linus Torvalds)에 의해 만들어진 깃은 소프트웨어 프로젝트뿐만 아니라 어떤 파일에든 변경사항을 추적하기 위해 적용될 수 있으며, Gitkraken 등 깃을 활용하여 재배포된 수많은 소프트웨어가 존재한다.

[4] https://guides.github.com/introduction/flow/

브랜치는 마스터에서 뻗어 나온 가지와 같아서 마스터의 코드를 모두 담고 있지만, 관리자에 의해 병합되지 않는 이상 브랜치 상의 코드 변화는 마스터에 영향을 미치지 않는다.

혹은 프로젝트를 포크(Fork)하여 작업하는 방법도 있다. 포크란 해당 프로젝트를 복사하는 것을 말한다. 복사본에서는 마음껏 코드를 수정해도 원본 프로젝트에 영향을 미치지 않기 때문에, 수정한 코드를 테스트해보고자 할 때 유용하다.

(2) 두번째 단계는 코드를 작성하는 것이다. 이는 기존 코드의 오류를 제거하는 내용일 수도 있고, 아예 새로운 기능을 추가하는 코드일 수도 있다.

(3) 세번째 단계에서는 이렇게 수정한 코드 조각을 간단한 설명과 함께 보여주는 풀 리퀘스트(Pull Request)를 작성한다. 즉 풀 리퀘스트란 프로젝트 관리자에게 위에서 수정한 소스 코드의 원본 코드에 대한 병합을 요청하는 글이다.

(4) 다음으로는 풀 리퀘스트에 대해 상호 검토가 이루어진다. 누구나 풀 리퀘스트에 대한 평가를 댓글을 남길 수 있으며 관련된 이슈(Issue)를 제기할 수도 있다. 이슈는 깃허브의 커뮤니케이션 채널로, 누구나 프로젝트에 대한 의견을 제시할 수 있는 일종의 게시판이다.

(5) 다섯 번째 단계는 프로젝트 관리자에 의해 풀 리퀘스트의 수용 여부가 결정되는 단계이다. 풀 리퀘스트가 거부될 경우 해당 코드는 프로젝트로 병합되지 않고 반려된다. 하지만 풀 리퀘스트가 수용될 경우 해당 코드는 프로젝트의 커밋으로 생성된다. 물론 관리자의 재량에 따라 마스터로 바로 병합될 수도 있지만, 보통은 먼저 브랜치 상에서 커밋을 생성한다.

(6) 그리고 마지막 단계에 비로소 브랜치가 마스터로 병합(merge)된다.

위와 같은 일련의 과정을 통해 깃은 불특정 다수의 협업이 효율적으로 이루어질 수 있도록 돕는다. 이러한 과정을 크게 두 부분으로 나누어 본다면, [그림 2]와 같은 모습이 된다. 풀 리퀘스트를 통해 커밋을 제한적으로 생성하는 첫 번째 규제 장치가 [A] 부분이라면, 브랜치를 통해 수정사항이 곧바로 마스터에 반영되는 것을 방지하는 장치가 [B] 부분에 해당된다. 물론 OSS 프로젝트의 모든 커밋이 이러한 과정을 온전히 거치고 생산되는 것은 아니다. 커밋을 직접 추가할 수 있는 권한을 가진 프로젝트 관리자들은 때때로 풀 리퀘스트를 통해 심사를 받지 않고

직접 OSS 의 코드를 수정하기도 한다. 따라서 OSS 프로젝트의 전체 커밋 정보는 풀 리퀘스트 중 공개적인 심사를 통과한 것들과, 커밋 권한자가 직접 추가한 커밋 두 가지 유형으로 구분될 수 있다(원인호, 2014).

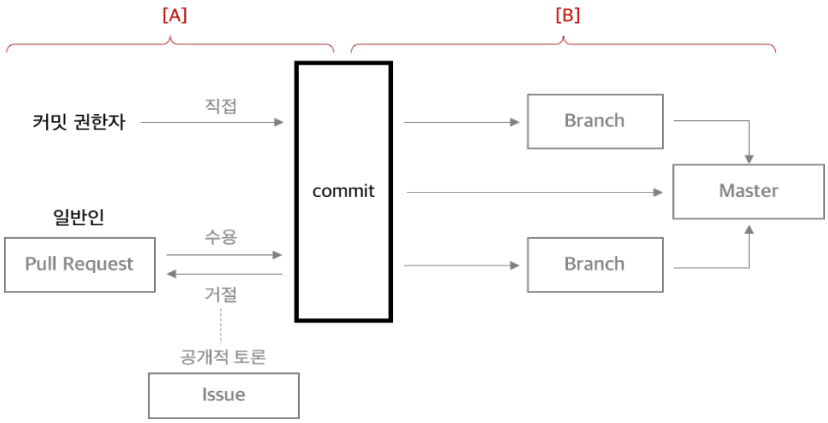

그림 2. 코드 병합(Merge)의 과정

3.2. 데이터 수집

본 연구에서는 OSS 프로젝트의 효율성을 논함에 있어 이러한 커밋이 생성되고 마스터 코드로 병합되는 과정을 반영하고자 한다. 이를 위해 깃에 기반한 개발 프로세스에서 등장하는 커밋, 브랜치, 포크, 풀 리퀘스트, 이슈 등 다양한 기능에 대한 데이터가 필요한데, 이는 깃허브 API(Application Programming Interface) 를 통해 수집이 가능하다. 깃허브에서는 무료 API 서비스를 제공함으로써 외부로부터의 축적된 각종 데이터에 대한 접근을 허용하고 있다.

본 연구에 필요한 DMU 정보 수집은 깃허브에서 제공하는 쇼케이스 페이지의 분류를 기준으로 이루어졌고, 그 외 투입 및 산출 변수와 환경 변수에 해당되는 값을 수집하는 데는 깃허브 API 와 Spreadseet 등의 젬(gem)을 이용한 크롤링이 동원되었다.[5]

[5] 지난 5 월 22 일 깃허브는 GraphQL 을 이용한 4 번째 버전 API 를 공개하였으나, 일부 데이터를 가져오기에 적합하지 않아 본 연구에서는 기존의 REST API 를 이용하였다.

4. 연구 모형

4.1. DEA

Data Envelope Analysis(DEA)란 다수의 투입물과 다수의 산출물을 가진 의사결정단위(Decision Making Units, DMUs)의 상대적 효율성을 측정하는 기법이다. 소프트웨어 프로젝트의 효율성을 측정하는 데는 보통 DEA 를 이용한다. 특히 OSS 의 경우 생산 과정이 정의하기 어렵고 복합적이므로, 생산 함수를 명시적으로 가정할 필요 없이 경험적 데이터를 이용하여 효율적 프론티어를 도출하고 이로부터의 거리를 토대로 각 평가 대상의 효율성을 측정하는 DEA 가 매우 유용하다. 또한 DEA 는 어떤 효율적인 DMU 를 참조하여 어떤 투입 또는 산출 요소에서 얼마만큼의 개선이 필요한지와 같은 성과 벤치마킹에 유용한 정보를 제공한다.

이러한 DEA 를 활용하여 효율성 분석을 하기 위해서는 기본적으로 DEA 모형의 규모에 대한 수익(Returns to Scale) 가정을 설정해야 하는데, OSS 프로젝트의 경우 규모가 각양각색이며 IT 산업은 규모에 대해 수익이 증가하는 경우와 감소하는 경우가 모두 존재하므로 규모수익가변의 가정이 적용된다(Kitchenham, 2002). 따라서 본 연구는 규모수익가변을 가정하는 BCC 모형을 채택한다.

4.2. Two Stage DEA

깃허브의 OSS 프로젝트는 매우 복합적인 개발 프로세스를 동반한다. 통상적인 DEA 는 이러한 단계적인 커밋 생산 과정을 하나의 블랙박스처럼 간주하여, 깃허브 OSS 프로젝트의 효율성을 정확히 담아내지 못한다. 따라서 본 연구는 DMU 의 내부 구조를 단일 단계가 아닌 2 단계로 세분화하여 Two Stage DEA 모델을 활용해 효율성을 분석하려 한다. Two Stage DEA 를 적용하면, 전통적 DEA 모형에서는 효율적으로 나타나던 DMU 도 실제로는 내부적으로

비효율을 내포하고 있음을 보여줄 수 있다. 또한 각 단계의 효율성을 독립적으로 계산할 수 있기 때문에, 구체적으로 어느 단계가 비효율이 초래하는지 알려주기 때문에, 효율성을 증진하기 위한 가이드라인을 단계별로 구체화할 수 있다.

깃허브에 기반한 OSS 개발 과정에서 '커밋'은 상징적인 의미를 가진다. 커밋이란 앞서 언급했듯이 새로운 코드의 작성이 이루어질 때마다 깃에 의해 생성된 해당 부분에 대한 기록을 담고 있는 독립적인 데이터 단위를 가리킨다(원인호, 2014). 깃허브는 기본적으로 이러한 커밋 정보를 편리하게 관리하고 공유하는 것을 지원하는 서비스라고 해도 과언이 아니다. OSS 를 구성하는 모든 코드가 적어도 하나의 커밋에 담겨 있기 때문이다. 본 연구에서는 OSS 프로젝트의 효율성을 논함에 있어 이러한 커밋이 생성되고 마스터 코드로 병합되는 과정을 반영하고자, Two Stage DEA 모델의 중간산출물로 '커밋의 수'를 설정하였다. 이를 바탕으로 1 단계의 Merge Efficiency 는 다수의 참여자가 얼마나 효율적으로 커밋을 생산할 수 있는지 평가하였고, 2 단계의 Project Efficiency 는 그렇게 생산된 커밋이 얼마나 효율적으로 질적, 양적 결과물을 산출했는지 평가하였다.

(1) 1 단계 – Merge Efficiency

OSS 프로젝트의 전체 커밋은 풀 리퀘스트 중 단계적 규제를 통과한 것과, 커밋 권한자가 직접 추가한 커밋 두 가지 유형으로 구분된다. 따라서 1 단계의 Merge Efficiency 는 풀 리퀘스트의 수와 커밋 권한자 수를 투입으로 하며 커밋의 수를 산출로 한다. 이때 OSS 프로젝트의 경우 따로 비용을 지불하고 인적 자원을 고용하는 것이 아니라 불특정 다수가 자발적으로 참여하는 형태이므로 투입에 대한 통제력은 없다고 볼 수 있다. 따라서 주어진 투입에 대하여 산출의 극대화를 추구하는 산출 지향 모델이 적합하다(Koch, 2008).

(2) 2 단계 – Project Efficiency

2 단계에서는 커밋의 수를 투입으로 하여 좋아요(Star)의 수와 프로젝트 크기를 산출로 효율성을 분석한다. 이때 분석되는 Project Efficiency 는 산출에 비해 투입이 상대적으로 고정적이며, 커밋의 수가 주어진 상태에서 산출물의 극대화를 추구하기 때문에 산출지향 모델을 채택한다.

4.2. DMU 선정

OSS 효율성 연구에 DEA 모형을 제대로 적용하기 위해서는 동질적인 DMU 간에 비교가 이루어져야 한다. 동질성이 떨어질 경우 분석 대상이 아닌 외부적 요인에 의해 결과가 좌우될 수 있기 때문이다(Sarkis, 2007). 이를 위해 Ghapanchi & Aurum(2012b)은 소스포지가 제공하는 카테고리 상에서 '소프트웨어 개발'로 분류되는 프로젝트로 표본의 범위를 좁혔다. 이외에도 대부분의 선행 연구에서 ERP 프로젝트(Stensrud & Myrtveit, 2003), Y2K 프로젝트(Yand & Paradi, 2004), 보안 관련 프로젝트(Wray & Mathieu, 2008) 등의 주제로 DMU 를 한정하였다.

깃허브의 경우 소스포지와는 달리 주제별 카테고리를 가지고 있지 않지만, 쇼케이스 페이지를 통해 일부 인기 있는 주제의 프로젝트 묶음을 제공한다. 본 연구에서는 깃허브에서 제공하는 총 52 가지 주제 중 다음 두 가지 조건을 만족하는 프로젝트가 가장 많은 '웹 프레임워크(Web Application Frameworks)' 묶음을 대상으로 하여 동질성을 확보하고자 한다.

(1) 라이선스

개별 OSS 에 대한 소유와 배포에 대한 권한은 다양한 라이선스를 통해 구체화된다. 본 연구에서는 공식적으로 OSS 에 해당되는 프로젝트로 분석 대상을 한정하고자, License.md 또는 License.txt 파일이 명시적으로 포함된 프로젝트만을 선별하였다.

(2) 활동성

연구의 실효성을 보장하기 위해 최근까지 활발한 활동이 이루어졌으며 깃의 기능을 온전히 사용하고 있는 프로젝트만을 대상으로 한다. 최소 1 개월 이내에 커밋이 제출되었으며 최소 20 명의 기여자가 참여하고 풀리퀘스트와 이슈가 존재하는 프로젝트를 선정하였다.

깃허브에서는 서버사이드 개발의 골조를 제공하는 '웹 프레임워크'를 주제로 총 29 개의 프로젝트를 소개하고 있다. 29 개 프로젝트 모두가 위 두 조건을 만족하여 DMU 로 선정되었다. 이는 사용되는 변수 개수의 세 배보다 큰 값이므로 유효한 DMU 수이다(Cooper et al., 2000).

4.3. 투입 및 산출 선정

OSS 프로젝트의 효율성을 측정할 때 투입으로 '개발자의 수'를 설정하곤 한다(Ghapanchi, 2012a). 그러나 단순히 깃허브에서 제공하는 기여자(Contributor)의 수만을 투입 요소로 삼는 것은 논리적 오류가 있다. 넓은 의미에서 OSS 프로젝트에의 '기여자'는 코드 작성자와 더불어 이슈를 통해 버그를 제보하거나 의견을 제시한 사람까지 포괄한다. 그러나 깃허브에서 정의한 기여자의 개념은 커밋 권한자로 한정된다. 이는 깃허브가 커밋의 수를 기준으로 기여도를 부여하기 때문이다. [그림 2]의 [A]를 참고하면 전체 커밋은 커밋 권한자와 풀 리퀘스트 두 가지 경로로 생성된다. 따라서 OSS 프로젝트 효율성의 투입으로는 *기여자의 수*와 더불어 *풀 리퀘스트의 수*를 선정하였다. 그러나 커밋 권한자 즉 기여자가 항상 풀 리퀘스트를 거치지 않고 커밋을 보내지 않기 때문에 투입 요소 간 중복이 있을 수 있다. 상관 분석 결과에서도 두 변수가 낮은 수준이지만 양의 상관 관계를 어느 정도 가지므로, 다중공선성의 문제가 발생할 수 있다는 한계를 가진다.

	NumberOfContributors	NumOfPullRequests
NumberOfContributors	1	
NumOfPullRequests	0.631738	1

표 2. 투입 간 상관분석

중간 변수로 선정한 커밋의 수는 유의미한 것으로 평가되어 선별된 코드를 가리키므로 수용되지 않은 코드까지 포함하는 투입과 구별된다. 또한 커밋은 새로운 코드의 추가뿐만 아니라 잘못되거나 비효율적인 코드의 삭제를 담고 있기도 한다. 따라서 프로젝트의 양적 산출을 측정하는 프로젝트 크기(Project Size)와도 극명히 구별되는 개념이다.

산출의 경우, OSS 프로젝트로 생성되는 결과물의 양적 지표로는 *프로젝트 파일의 크기*를 채택하였다(Koch, 2009). 그러나 양적 지표는 OSS 의 성격을 온전히 반영하지 못하며 산출의 일면만을 보여준다. 이에 Wrau & Mathieu (2008)은 프로젝트 순위를 질적 산출물로 제안하였다. 깃허브의 경우 OSS 프로젝트의 랭킹을 따로 제공하지는 않지만, 즐겨찾기(Star)의 수와 팔로우(Watch)의 수를 기준으로 정렬할 수 있다. 즐겨찾기에 등록한 프로젝트는 메뉴의

즐겨찾기 란에 모아져 있으며, 팔로우한 프로젝트는 대시보드로 관련 알림을 보내주는 기능이다. 두 기능 모두 OSS 프로젝트의 품질에 대한 평가가 내포된 수치이기 때문에 질적 산출물로 적절하다고 보았다. [표 3]은 산출 간의 상관분석 결과인데, 즐겨찾기 수와 팔로우 수의 상관관계가 약 0.8579 로 상당히 높다. 따라서 본 연구에서는 양적 산출인 *프로젝트 크기*와 함께, 유저가 보다 쉽게 접근할 수 있는 기능인 *즐겨찾기의 개수*를 산출물로 설정하고 팔로우의 개수는 산출에서 배제했다.

	Project Size	Number of Stars	Number of Watches
Project Size	1		
Number of Stars	0.14855316	1	
Numbef of Watches	0.15674469	0.85791255	1

표 3. 산출 간 상관분석

.

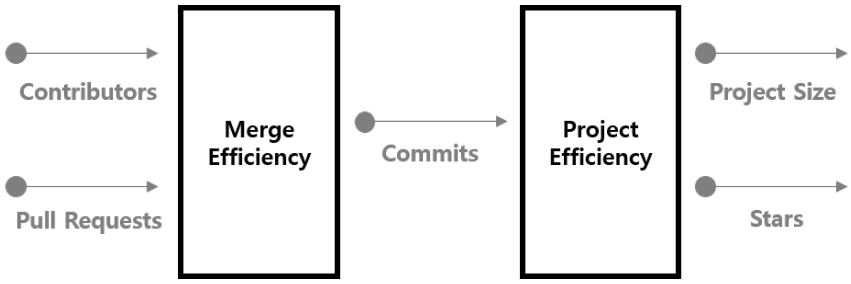

그림 3. Two Stage DEA 를 활용한 연구 모형

5. 결과 분석

5.1. 1 단계 – Merge Efficiency

구분	효율성	효율성	산출부족분	투영점	준거 및 참조	
DMU	CRS	VRS	Y1	Y1	준거집단	참조횟수
meteor	0.4822	1	0	18749	1	12
rails	1	1	0	62632	2	5
laravel	0.1163	0.2569	15508.752	20871.752	1,14	0
express	0.2912	0.4373	6924.834	12306.834	1,23	0
flask	0.0956	0.2056	11311.857	14238.857	1,23	0
django	0.4506	0.6466	13344.874	37757.874	2,14	0
sails	0.3438	0.5276	5922.928	12538.928	1,23	0
symfony	0.5294	0.6064	20318.942	51628.942	2,14	0
spring-framework	0.5942	0.9569	664.784	15415.784	1,14,23	0
mean	0.1129	0.1541	10050.852	11881.852	1,23	0
playframework	0.1643	0.2986	19386.406	27639.406	1,14	0
sinatra	0.1765	0.3171	8184.569	11984.569	1,23	0
revel	0.1241	0.1302	9046.574	10400.574	23,29	0
cakephp	0.7023	1	0	32569	14	8
generator-jhipster	0.2847	0.5242	9862.113	20727.113	1,14,23	0
Nancy	0.2017	0.3697	9086.848	14416.848	1,23	0
nodal	0.2831	0.328	1575.771	2344.771	2,19,23	0
derby	0.613	0.6756	827.057	2549.057	19,23,29	0
whitestorm.js	0.8591	1	0	1744	19	6
hanami	0.1652	0.172	6795.926	8207.926	23,29	0
padrino-framework	0.2821	0.4326	7165.884	12629.884	1,23	0
aspnetboilerplate	0.5592	0.5693	2448.926	5685.926	19,23,29	0
mojo	1	1	0	10908	23	18
ninja	0.2766	0.2859	4825.71	6758.71	2,19,23	0
kemal	0.147	0.1555	3031.334	3589.334	19,23,29	0
web2py	0.4709	0.5454	5977.949	13149.949	1,14,23	0
pakyow	0.6029	0.7804	383	1744	19	0
frappe	0.7504	0.7625	3926.326	16527.326	2,14,23	0
catalyst-runtime	1	1	0	4493	29	5

총 29 개의 OSS 프로젝트의 병합 효율성(Merge Efficiency)을 산출지향 BCC 모델로 DEA 분석을 실시하였다. 투입 요소는 기여자의 수와 풀 리퀘스트의 수, 현 단계에서 산출이자 전체 모형의 중간 변수는 커밋의 수(Y1)이다.

VRS 효율성을 기준으로 29 개의 프로젝트 중 6 개의 프로젝트가 효율적이며 효율성이 1 이다. 상대적으로 효율성이 낮은 프로젝트는 효율성의 값이 1 보다 작다. 준거 및 참조 열에 주어진 준거집단 자료와 참조 횟수는, 각각 비효율적인 DMU 에 대하여 효율성의 척도로 제시되는 벤치마킹 대상과 다른 DMU 에게 벤치마킹 대상으로 참조된 횟수를 가리킨다. Merge Efficiency 에 있어서는 Mojo 와 meteor 가 각각 18 회, 12 회로 가장 많이 준거 기준이 되었고, cakephp 와 rails, catalyst-runtime 이 그 뒤를 따른다.

5.2. 2 단계 – Project Efficiency

구분	효율성	효율성	산출부족분	산출부족분	투영점	투영점	준거 및 참조	
DMU	CRS	VRS	Y1	Y2	Y1	Y2	준거집단	참조횟수
meteor	0.2289	1	0	0	67855	37517	1	13
rails	0.071	1	0	0	146825	35750	2	7
laravel	0.6479	1	0	0	8563	32274	3	8
express	0.6407	0.992	59.543	259.033	8681.543	32279.033	1,3,19	0
flask	1	1	0	0	4624	27291	5	8
django	0.1423	0.9926	1220.522	195.024	165318.522	26112.024	1,2,19	0
sails	0.2839	0.5343	10056.401	14958.264	21603.401	32119.264	1,3,19	0
symfony	0.063	0.5552	77731.091	11658.52	174757.091	26214.52	1,2,19	0
spring-framework	0.1323	0.5975	68153.278	9564.7	169318.278	23761.7	1,2,19	0
mean	0.6012	0.6543	3520.564	5317.252	10211.564	15379.252	5,17,19	0
playframework	0.1665	0.4779	94823.088	10228.94	181617.088	19593.94	1,3,19	0
sinatra	0.2686	0.3307	12287.866	18892.354	18353.866	28226.354	3,5,19	0
revel	0.6601	0.7871	637.302	2243.032	2944.302	10540.032	5,17,19	0
cakephp	0.0327	0.3313	143009.194	14340.157	213866.194	21445.157	1,2,19	0
generator-jhipster	0.0776	0.2376	66319.186	22696.612	86989.186	29770.612	1,3,19	0
Nancy	0.142	0.2773	117077.073	13752.152	161995.073	19028.152	1,3,19	0
nodal	0.6094	1	0	0	1243	4349	17	6
derby	0.27	0.2985	5939.06	10013.476	8471.06	14275.476	5,17,19	0
whitestorm.js	1	1	0	0	304616	3959	19	22
hanami	0.3756	0.4306	33148.564	5232.87	58212.564	9190.87	5,17,19	0
padrino-framework	0.0678	0.1198	79127.997	22304.609	89900.997	25341.609	1,3,19	0
aspnetboilerplate	0.1221	0.1551	129439.78	14926.94	153209.78	17667.94	3,5,19	0
mojo	0.0273	0.1088	194344.026	14961.117	218067.026	16787.117	1,2,19	0
ninja	0.1808	0.183	174667.035	7529.749	213785.035	9215.749	5,17,19	0
kemal	0.3127	1	0	0	451	1620	25	0
web2py	0.0441	0.1438	229045.323	8342.119	267518.323	9743.119	1,2,19	0
pakyow	0.0665	0.0782	27848.634	8887.761	30211.634	9641.761	5,17,19	0
frappe	0.0457	0.3301	204049	3351	304616	3959	19	0

catalyst-runtime	0.0187	0.0449	282982.276	5301.145	296276.276	5550.145	1,2,19	0

프로젝트 효율성의 경우 VRS 를 기준으로 총 29 개의 프로젝트에 대하여 7 개의 프로젝트가 효율적으로 나타났다. 놀라운 것은 두 단계 모두 1 의 효율성을 띠는 meteor, rails, whitestorm 를 제외하면, 다른 네 프로젝트 nodal, laravel, flask, kemal 은 Merge Efficiency 에 있어서는 극명한 하위권에 속했다는 점이다. 즉 네 프로젝트의 종합적인 효율성을 극대화하기 위해서는 Merge Efficiency 의 개선이 이루어져야 한다는 점을 시사한다. 반대로 Merge Efficiency는 높은 반면 Project Efficiency가 떨어지는 cakephp, mojo, catalyst-runtime의 경우도 마찬가지이다. 이는 단계를 세분화하여 효율성을 측정하지 않고 전통적 DEA 모형에 머물렀다면 알 수 없는 내용이다.

5.3. 최종 효율성과 블랙박스 효율성

DMU	1 단계 효율성	2 단계 효율성	최종 효율성	블랙박스 효율성
meteor	1	1	1	1
rails	1	1	1	1
laravel	0.2569	1	0.2569	0.8603
express	0.4373	0.992	0.433802	1
flask	0.2056	1	0.2056	0.8106
django	0.6466	0.9926	0.641815	0.9626
sails	0.5276	0.5343	0.281897	0.5461
symfony	0.6064	0.5552	0.336673	0.522
spring-framework	0.9569	0.5975	0.571748	0.6036
mean	0.1541	0.6543	0.100828	0.371
playframework	0.2986	0.4779	0.142701	0.4404
sinatra	0.3171	0.3307	0.104865	0.3166
revel	0.1302	0.7871	0.10248	0.4631
cakephp	1	0.3313	0.3313	0.3424
generator-jhipster	0.5242	0.2376	0.12455	0.21
Nancy	0.3697	0.2773	0.102518	0.2467
nodal	0.328	1	0.328	0.9251
derby	0.6756	0.2985	0.201667	0.8527
whitestorm.js	1	1	1	1
hanami	0.172	0.4306	0.074063	0.2815
padrino-framework	0.4326	0.1198	0.051825	0.1162
aspnetboilerplate	0.5693	0.1551	0.088298	0.2764
mojo	1	0.1088	0.1088	0.1227
ninja	0.2859	0.183	0.05232	0.1654
kemal	0.1555	1	0.1555	0.2342
web2py	0.5454	0.1438	0.078429	0.1443
pakyow	0.7804	0.0782	0.061027	0.1905

frappe	0.7625	0.3301	0.251701	0.3301
catalyst-runtime	1	0.0449	0.0449	0.0558

Two Stage DEA 모델에서 각 단계의 결과를 종합하여 최종 효율성을 도출하는 방식에는 크게 가산법과 곱셈법 두 가지가 있다. 가산법의 경우 가중평균을 적용하여 최종 값을 계산하는데, 이때 가중치의 계산 방식이 굉장히 다양하며 이를 OSS 프로젝트의 성격에 맞게 적용하기란 상당히 어려운 일이다. 따라서 추정치에 불과한 가중평균을 사용하는 것이 오히려 분석의 정확도를 떨어뜨릴 수 있다고 보아, Kao(2008)에서 소개한 곱셈형 방법을 통해 최종 효율성을 도출하였다.

또한 전통적 DEA 방식에 따라 중간 단계를 블랙박스로 간주하고 최초 투입과 최종 산출만으로 계산한 블랙박스 효율성을 구하였다. 이를 최종 효율성과 비교하면, Two Stage DEA 를 통해 분석한 효율성이 블랙박스 효율성보다 낮거나 같음을 알 수 있다. 이는 Two Stage DEA 모형이 OSS 프로젝트의 개발 프로세스를 보다 세분화하여 내부적 비효율성까지 포착하기 때문이다. 이는 대표적으로 Express 의 사례를 통해 재확인할 수 있다. 전통적 모델에 따르면 효율성 값이 1 이지만, 최종 효율성이나 세부 단계 효율성을 살펴보면 모두 내부적 비효율성이 반영되어 있다.

5.3. Kruskal-Wallis 검정

"보는 눈이 많아지면, 모든 버그가 드러난다."
일명 리누스 법칙(Linus' Law)이라 일컬어지는 Raymond(1999)의 한 구절이다. 이는 오픈 소스의 핵심 원리를 '다수의 협업'으로 해석한다. OSS 는 불특정 다수의 기여를 통해 자율적 협업이 이루어지는 언뜻 듣기에는 이상적인 이야기이다. 실제로 OSS 개발은 자유와 평등을 중시하고, 대중의 기여와 공유를 통해 이루어져 상업용 소프트웨어의 독점 현상에 반기를 들며 시작되었다.

그러나 때로 다수의 참여는 때로 독이 될 수 있다. 집단의 규모가 커질수록 비효율적 의사소통이 발생할 가능성이 늘어나고, 프로젝트의 원활한 관리가 어려워지기 때문이다. 원인호(2014)에 따르면 효율적인 OSS 프로젝트를 위해 필요한 것은 일반 참여자의 단순 증가가 아닌, 오히려 엘리트 참여자의 증가와 다수에 대한 소수의 통제이다.

이에 '다수의 참여'가 Two Stage DEA 중 1 단계를 통해 도출된 병합 효율성(Merge Efficiency)의 값과 가지는 관계를 알아보기 위하여 간단한 두 가지 가설에 대하여 Kruskal-Wallis Test 를 시행하였다. Kruskal-Wallis Test 는 Mann-Whitney U-test 의 확장 모델로 3 개 이상의 그룹에 대한 분석이 가능하다. 검정을 실시한 결과는 다음과 같다.

가설	Sig.	의사결정
전체 Commit 수 에 대한 Contributor 수의 비율이 낮을수록 Merge Efficiency 는 높을 것이다.	.017	채택 (귀무가설 기각)
Issue 의 수가 많을수록 Merge Efficiency 는 높을 것이다.	.108	기각 (귀무가설 유지)

첫 번째 가설에 따르면, 전체 커밋 대비 풀 리퀘스트의 비율이 낮을수록 효율적이다. 이는 풀 리퀘스트를 거치지 않고 커밋 권한자에 의해 직접적으로 생성된 커밋이 오히려 효율성에 긍정적으로 작용함을 뜻한다. 즉 공개적인 다수의 참여 방식 이외에도 소수의 비공개적인 생산방식이 OSS 프로젝트의 효율성을 더 높일 수 있는 것이다. 이를 통해 프로젝트 효율성은 높지만 병합 효율성은 극도로 낮은 nodal, laravel, flask, kemal 와 같은 DMU 들이 커밋 대비 풀 리퀘스트의 비율을 낮추는 방안을 고려할 하다는 결론을 얻을 수 있다.

두 번째 가설에 의하면, 이슈의 수가 많을수록 OSS 프로젝트는 비효율적이다. 이는 지나친 다수의 참여는 오히려 의사소통의 어려움을 가져와 프로젝트에 부정적 영향을 미칠 수 있음을 시사한다.

6. 결론 및 한계

본 연구에서는 오픈소스 소프트웨어 프로젝트의 효율성을 상업용 소프트웨어와는 다른 시각으로 바라보는 데 더하여, 중간 변수의 도입을 통해 효율성의 개념에 깃(git)에 기반한 OSS 의 내부처리과정을 새롭게 반영하였다. 또한 도출된 효율성과 가설 검정 결과를 바탕으로 다수의 균등한 참여 증가가 적절히 통제되지 않는 상황은 OSS 프로젝트 효율성에 부정적 영향을 미친다는 결론을 확인하였다.

한편 본 연구는 다음과 같은 한계를 가진다.

첫째는 데이터 수집의 한계이다. 예컨대 산출 변수 프로젝트 크기(Project Size)는 개발자의 입장에서 최소화하고자 하는 수치이다. 이는 산출의 양적 결과값을 최대화하고자 하는 전통적 관점과 상충된다. Wray & Mathieu (2008)는 이러한 상충 관계를 해결하기 위해 새로운 산출 변수로 프로젝트 용량을 다운로드 횟수로 나눈 값인 '유용한 파일의 크기'를 제시하였다. 그러나 이는 다운로드가 중요한 기능 중 하나였던 소스포지 위주의 변수이다. 깃허브를 기준으로 보면, 다운로드 횟수는 큰 의미를 갖지 못한다. 유저가 OSS 를 이용할 때 다운로드보다도 Clone 이나 Fork 와 같은 기능을 사용하기 때문이다. 나아가 깃허브에서 제공하는 API 로도 수집할 수 없는 데이터에 해당한다. 또한

둘째는 연구 방법론의 한계이다. 본 연구에서 사용한 Two Stage DEA 는 두 개의 개별 단계들을 독립적인 DMU 로 간주하여 효율성을 별개로 측정하는 표준 접근법이다. 표준적인 Two Stage DEA 는 단순하여 사용이 용이하지만, 중간 변수로 인해 두 단계 간의 상충 현상을 초래할 수도 있다. 예컨대 첫 번째 단계의 효율성을 높이기 위해 첫 번째 단계의 산출 즉 전체 모형의 중간 변수 양을 늘려야 하는데, 이는 두 번째 단계 투입 양의 증가로 이어져 두 번째 단계의 효율성이 낮아지는 현상이 초래된다. 이러한 한계를 해결하기 위해 Kao and Hwang(2008)가 개발한 Two Stage Network DEA 모형 등을 이용하면 2 단계 프로세스의 효율성을 보다 타당하게 측정할 수 있었을 것이다.

7. 참고 문헌

이남형 & 김준일. (2015). 경제학이 오픈소스 소프트웨어에 대해 알고 있는 것: 역사, 산업 조직, 정책 함의. 경제학연구, 64(2), pp.127-166.

원인호. (2014). 소셜 코딩 서비스 '깃허브'를 통한 오픈소스 소프트웨어 공동체의 협력과 규제. 서울대학교 대학원.

Banker, R., Charnes, A. & Cooper, W. W. (1984) Some models for estimating technical and scale.

Charnes, A. & Cooper, W. W. (1962). Programming with linear fractional functionals. Naval Research Logistics Quarterly 9, 181-185.

Ghapanchi, A. H. & Aurum, A. (2012). The impact of project capabilities on project performance: Case of open source software projects. International Journal of Project Management, 30(4), pp.407-417.

Ghapanchi, A. H. & Aurum, A. (2012). Competency rallying in electronic markets: implications for open source project success. Electronic Markets, 22(2), pp.117-127.

Ghapanchi, A. H. (2011). A taxonomy for measuring the success of open source software projects. First Monday, 16(8).

Koch, S. (2008). Exploring the effects of SourceForge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis. Springer Science.

Sarkis, J. (2007). Preparing your data for DEA. Modeling Data Irregularities and Structural Complexities in Data Envelopment Analysis, pp.305-320.

Wray, B. A. & Mathieu, R. G. (2007). The Application of DEA to Measure the Efficiency of Open Source Security Tool Production. AMCIS 2007 Proceedings, p.118.

Wray, B. A. & Mathieu, R. G. (2008). Evaluating the performance of open source software projects using data envelopment analysis. Information Management & Computer Security, 16(5), pp.449-462.