國立政治大學資訊科學系
Department of Computer Science
National Chengchi University

碩士論文

Master's Thesis

軟體典藏庫資料分析：以GitHub為例

Data Analysis for Software Repository：A Case Study of GitHub

研 究 生：劉耀文

指導教授：徐國偉

中華民國一零四年十月

October 2015

軟體典藏庫資料分析：以GitHub為例

Data Analysis for Software Repository：

A Case Study of GitHub

研 究 生：劉耀文　　Student：Yao-Wen Liu

指導教授：徐國偉　　Advisor：Kuo-Wei Hsu

國立政治大學

資訊科學系

碩士論文

A Thesis

submitted to Department of Computer Science

National Chengchi University

in partial fulfillment of the Requirements

for the degree of

Master

in

Computer Science

中華民國一零四年十月

October 2015

致謝

在研究所的這兩年期間，對我來說意義非常重大，選擇念在職專班更是我人生中相當重要的一個決定。時光匆匆，兩年的學習歷程也到了收尾的階段。首先，要感謝的是我的指導教授徐國偉老師，給了我很大的空間去思考，讓我在整個研究的過程中能思考得更為透徹，學習到了獨立思考的能力，並且老師能在適當的時候提出建議，也在對談的過程當中學習到了思考的邏輯性，讓研究的過程中更為順利。

另外要感謝的是兩位口試委員，劉吉軒老師與黃信貿老師，兩位在口試時提出實際且寶貴的建議，後續的修改讓整篇論文能更具有架構性與說服力，使論文能更為完整。同時也要謝謝一起學習與陪伴的同學們，從學習期間的作業與報告，到論文研究，都與同學們互相學習與幫助，也讓我在研究所生活過得更為充實與豐富，同學們的互相加油打氣，也成為了我們完成論文的動力

最後，要感謝的是我的家人，謝謝他們的支持與鼓勵，是讓我繼續念研究所的動力，也希望能與他們一起分享畢業的喜悅。念在職專班的兩年期間，遇到了很多困難，不論是學業與工作上的平衡，時間上的調配與管理，體力上的瓶頸等等，對於一般人來說是難以體會的經驗，但是看見老師的認真教導，同學們的努力學習，便時時提醒著自己要跟上其他人的腳步。如今兩年的試煉也到了尾聲，我想這不會是個結束，反而是個新的開始，期望能在老師、同學與家人的陪伴與鼓勵下，繼續往下一階段前進，挑戰更高的目標，有句話這樣說著：「學然後知不足」，勉勵自己抱持著學習的態度去看待每一件事，才能知道自己該補足的部分，與大家共勉之。

劉耀文

民國 104 年 10 月

摘要

GitHub 是 2008 年開始發展，提供線上源碼託管服務的網路平台。除了提供使用者建立組織、專案和存取軟體庫之外，更提供一些網站社交功能，包括允許使用者追蹤其他用戶、加入專案組織與關注軟體庫的動態，並且對於軟體源碼的修改和針對程式錯誤(bug)提出評論等，使用者或組織成員透過平台上版本控管服務來共同開發軟體專案，並透過 GitHub 提供的社交服務來完成溝通與協調。

本研究針對 GitHub 資料集進行整體性的觀察與分析，透過不同的社群網絡指標與分析方法，發現 GitHub 平台上的協同合作與社交活動。舉例來說，為了找出 GitHub 上網路的彈性，我們使用度分布，還有近距中間度及參與中間度的值。同時，我們針對使用者或專案之間的互動情形來分析關聯性，並以平台上不同的操作事件來觀察使用者是否偏好某些行為，抑或是某些事件之間是否會互相影響。

研究目標希望能透過 GitHub 平台所取得的部分資料，來推論 GitHub 上的真實情況。希望透過專案之間的關聯性，來找出平台上最具影響力的專案或使用者；也將針對程式語言與公司組織的關聯性，觀察技術之間的可替代性，與公司之間的相互合作的情況。同時以 GitHub 平台上不同操作事件之間的相關性，觀察出何種操作行為會影響使用者進行貢獻(提交源碼)。

另一方面，本研究將以專案的吸引力與黏著度等角度，來針對 GitHub 平台上的專案進行分析。針對這兩種維度進行觀察，期望進一步得知專案的貢獻程度，與專案隨著時間的所產生的變化。換言之，本研究方法將針對資料集中所有專案進行演進的推論，區分出專案演進的四個階段(活躍期、流動期、穩定期、衰退期)，並分析出目前 GitHub 上專案所處於的階段，最後研究出各階段轉換所可能的機率為何，進一步推論出專案未來演進的趨勢。最後，本研究提出了其他延伸之議題，例如重新定義專案演進階段、選擇適合的專案成員與專案的推薦等，以提供未來可行的研究方向。

關鍵詞: 社群網絡、軟體典藏庫、資料分析

Abstract

GitHub began to develop in 2008, providing an online open source hosting platform. In addition to providing user-created organizations, projects and software repositories, it also provides more social features, including allowing users to track other users, join the dynamic project or organization, watch software repositories, modify the source code for the software, and make comments for the program error (bug).

In this study, we analyze of GitHub data sets; by using different network indicators and analysis methods in order to find collaboration and social activities on GitHub platform. For example, in order to find flexibility of networks on GitHub, we analyze degree distributions and values of closeness centrality as well as betweenness centrality. At the same time, we investigate the interaction between GitHub users and projects in order to analyze the correlation between them.

On the other hand, we analyze attraction and adhesion of the projects on GitHub platform. By using these two indicators, we can get the degree of contribution of the projects, and the changes of the projects over time. We consider the four stages of evolution (active, flow period, stable, recession) of the projects on GitHub. Finally, we study the probability of transition of the all stages, and further we infer the trend of the future evolution of the projects on GitHub.

Finally, this study could be extended and used to support other studies. For example, we can redefine the evolution stage of a project, select members for the project, and recommend the project.

Keywords: Social Network、Software Repository、Data Analysis

目錄

圖目錄

表目錄

第一章、導論

1.1 研究動機

隨著網路通訊的發展，目前使用者在網路上溝通與交流的形式越來越多，社群網站更是近幾年常被用來做為溝通的媒介。提到社群網站，如 Facebook、Google+、Twitter 等，都被視為是社群網站的首要指標，並以提供社群網絡服務（social networking service，SNS）為主，平台上透過擁有共同興趣、活動或工作領域的一群人所建立，這群人藉由這個虛擬平台進行聯繫與互動，因此，更構成了各式各樣的社群網絡，目前已有相當多的研究針對這樣形態的社群網站進行社群網絡分析。

GitHub[1]是 2008 年開始推出的線上源碼託管的網路平台，除了提供使用者建立組織、專案和存取軟體庫之外，更提供一些網站社交功能，包括允許使用者追蹤其他用戶、加入專案組織與關注軟體庫的動態，並且對於軟體源碼的修改和針對程式錯誤(bug) 提出評論等，使用者或組織成員透過網站版本控管來共同開發軟體專案，並透過 GitHub 提供的社交網絡服務來完成溝通與協調。

協同寫作[2]概念是由一群擁有共同興趣、目標或工作任務的組織，非個人獨立完成寫作計劃或專案。並由一位編輯者或編輯團隊監督與整合。讓此專案的格式或內容上，採用統一格式寫作，其他參與者能夠跟隨此專案既定規則寫作，讓文章前後較容易達成一致性。GitHub 平台上也透過協同寫作模式，利用專案擁有者所建立的規範，其他參與者遵守此規範來共同開發軟體，最後參與者可提交合併源碼要求(pull request)，並經由專案擁有者同意後，即可將程式源碼加入此軟體專案，並合併成新版本，發展出一種共同開發軟體專案的合作模式。因此，GitHub 同時結合了社群網絡與協同寫作兩大功能，也形成了一種不同於一般社群網站的網絡關係。

相較於一般性的社群網站，GitHub 的社交網絡關係更為複雜且多變，使用者牽

[1] http://zh.wikipedia.org/wiki/GitHub
[2] http://en.wikipedia.org/wiki/Collaborative_writing

涉的不僅是單純的溝通與交流，更涉及了協同開發軟體專案與程式版本管控等不同情況，使用者在平台上透過複製專案、建立分支版本、同步專案與提交合併專案要求，並於平台上討論與溝通專案異動與問題回覆等功能，以上因素都使 GitHub 產生更為複雜的網絡關係。

因此，透過分析 GitHub 平台上的網絡型態，進而觀察出使用者之間的操作行為模式，更能了解軟體開發人員在進行專案開發與合作時的交流過程，並了解平台上所提供的社交與協同寫作功能為軟體開發流程所帶來的效益，以助於往後在軟體開發品質的評估與效率的提升。

1.2 研究目標

本研究目標希望能透過 GitHub 平台所取得的部分資料，來推論 GitHub 上所呈現的真實情況。經由不同角度的分析，來觀察 GitHub 平台上使用者的互動與協同寫作的社交情形。希望透過專案之間的關聯性，來找出平台上最具影響力的專案或使用者，也將針對程式語言與公司組織的關聯性，觀察軟體開發技術之間的可替代性，與公司之間的相互投資或合作的情況。同時以 GitHub 平台上不同操作行為事件之間的相關性，觀察出何種操作行為會影響使用者最終的貢獻。以上均利用社群網絡分析的角度，期望能推論出以上論述。

另一方面，本研究目標將以專案的吸引力與黏著度等兩種角度，來針對 GitHub 平台上的專案進行分析。針對這兩種維度進行觀察，期望進一步得知專案的貢獻程度，與專案隨著時間的所產生的變化。換言之，本研究方法將以時間區段的概念來觀察專案的變化，期望能夠觀察出專案演進的過程，例如從專案建立的初期，到協同成員或貢獻者增加的成長期，到最後專案的穩定發展或衰退等階段，並且推算出各階段變化的機率，最後並推論出各個專案未來的走向與趨勢，以利 GitHub 使用者判斷此專案是否值得投入開發或學習。

1.3 論文成果

本研究首先針對資料集的可信度進行分析與驗證，透過與現實資料的比較，推論出本研究中所使用的資料具有一定程度可表達真實情況。並針對資料集中所有專案進行演進的推論，區分出專案演進的四個階段(活躍期、流動期、穩定期、衰退期)，並分析出目前 GitHub 上專案所處的階段，最後研究出各階段轉換所可能的機率為何，進一步推論出專案未來演進的趨勢，此論點為本論文最重要的貢獻之一。

本論文另一項重要貢獻為針對 GitHub 資料集進行整體性的觀察與分析，透過不同的社群網絡指標與分析方法，發現 GitHub 平台上的協同合作與社交活動之情形，本研究分析後所發現事項如以下:

(一) 針對資料集的可信度進行分析與驗證，推論資料集的可信程度

(二) 目前 GitHub 平台上使用者僅會針對單一使用者進行追隨，造成某一使用者擁有大量追隨者，不同於一般社群網站呈現相互追隨的情形，表示各使用者間的連結性是不足的，屬於低互動的社群網絡，並且從事件總數來看，使用者進行貢獻的次數為所有事件中最少，也表示目前 GitHub 上呈現低貢獻的情況

(三) GitHub 平台上呈現長尾效應現象，且最常使用的專案與使用者感興趣的專案並不一定成正相關

(四) 具有高度關聯性的專案通常是相同類型的程式語言

(五) 推論程式語言間的相關性與公司組織之間的關聯

(六) 追隨事件與最後是否進行貢獻之間並無顯著的關係，表示此兩種事件之間呈現微弱相關

(七) 推論 GitHub 專案的演進階段與演進機率

1.4 論文限制

本論文著重在 GitHub 平台上的資料，並以此平台所取得的資料為基礎，發展可行的

研究方向。因此，本論文對於資料的定義，例如專案吸引力與專案黏著度的定義敘述，與專案演進的四種型態定義等，僅適用於與 GitHub 相似，並具有協同寫作與社群功能之平台。

本論文針對社群網絡所觀察的角度，限制於 GitHub 平台的觸發事件，例如 watch、pull request 事件，若有其他平台相似於 GitHub 的觸發事件定義，仍為適用。本論文所發展的分析方法，例如專案貢獻度、專案吸引力與黏著度，以及專案演進等研究計算方法，僅適用於與 GitHub 有相同資料結構的平台；最後本論文的研究發現，僅限於與 GitHub 有相似功能與操作行為之平台。

1.5 論文章節架構

本論文以 GitHub 平台上的資料為基礎，研究主題圍繞在軟體庫的資料分析與社群網絡分析。第二章將介紹軟體庫資料探勘領域(MSR)[3]的背景介紹與發展，並且介紹本論文資料集的來源與取得方式，本章節後段說明本資料集的基本統計分析所發現之論點。第三章將介紹 GitHub 平台上的協同寫作與社群網絡分析，並以不同網絡分析指標來觀察本研究資料集中所發現的現象。第四章則以資料分析角度觀察本研究資料集中專案的影響力與重要程度，並介紹專案的吸引力與黏著度，與其專案貢獻程度；並以時間區段的概念來觀察專案發展的過程與演進，並探討專案與技術長期的趨勢走向。第五、六章則總結本研究的成果與發現，並說明未來可行的研究方向。

[3] http://2015.msrconf.org/

第二章、背景介紹

2.1 Mining Software Repositories(MSR)介紹

在 1960 年代中期軟體開發與維護的過程中遇到了嚴重的問題,導致軟體開發週期過長、費用昂貴、不符合需求與軟體品質低落等問題，也造成軟體專案延宕而無法順利完成，並有軟體危機[1]的一說。

因此，北大西洋公約組織（NATO）針對軟體開發時所遭遇困難，在 1968 年舉辦了首次軟體工程學術會議，在會中提出界定軟體開發所需相關知識，認為軟體開發的過程應該如工程般的嚴謹,「軟體工程」一詞因此產生。軟體工程自 1968 年正式提出後，一方面藉由學術界累積了大量的研究成果，也透過產業界進行技術的實踐，軟體工程已發展成為一門在軟體開發中不可缺少的專業領域。軟體工程的方法可細分成不同層面的涵義，包括專案管理、系統分析與設計、程式語言的編寫、系統測試、文件版本與開發品質控制[3]等，提供了一系列的標準和方法來指導團隊如何提升軟體開發過程的品質，而並非給出具體的開發過程的定義。

隨著軟體工程發展的演進，軟體工程研究人員也逐漸從軟體開發過程中尋找問題，並從記載的資料中挖掘訊息與知識，來觀察軟體演進的過程，進而產生了 MSR(Mining Software Repositories)[2]這項新的研究領域。MSR 以片面解釋是針對軟體庫(Software Repositories, SR)或程式源碼的資料進行資料探勘之行為[5]。而其中所謂軟體庫的定義所指的是在軟體開發或生產中所有演進過程所留下的資料與文件。它們的來源包括(一)metadata 的變更,如註解、使用者 id(或開發者 id)或時間戳記(time stamps)。(二)各版本之間的差異紀錄(變更紀錄或程式 log)的新增、修改、刪除。(三)軟體各版本間的分類。(四)bug 追蹤系統(bug report)。

而軟體庫中也記載了詳盡的軟體演進資料，例如:誰進行了修改、為何異動、何時完成等等。因此，MSR 則廣泛應用於 CVS 版本控管系統，來觀察軟體版本變更的過程。也就是說，MSR 並非使用資料探勘技術來研究調查軟體工程問題，而是利

用利用資料探勘或其他相關技術來研究調查軟體演進與變化。在 MSR 發展之前，也有針對長期軟體專案的數據來觀察軟體的演進，像是研究 IBM 產品 1969 至 2001 年間的變化，或觀察代碼衰變的現象，此類研究最為顯著的價值為研究出軟體演進的指標與理論。

由於開源軟體(open source)廣泛的發展，此類軟體開發模式也不斷的產生軟體數據資料[9]，也解決了過往 MSR 調查中缺乏歷史軟體數據的困境與阻礙，因此，MSR 領域的相關研究在近幾年也逐漸蓬勃發展。且 MSR 於每年舉行軟體探勘競賽(mining challenge)，針對當年度所釋出的資料集做為調查對象，來進行資料探勘之研究，並提出調查研究報告。

2.2 MSR 核心概念

以廣義而言，MSR 定義了兩類的問題，(一)Market-Basket Question（MBQ），指如果發生 A 事件，下次再發生是何時?是否為定期發生?(二)Prevalence Questions（PQ）是否有特定的功能增加/刪除/修改?有多少 function 被重新使用?以定義來說，最終分析的方法就是要達到上面這兩種目的。傳統軟體工程或其他相關領域的調查方法，已經被廣泛應用到 MSR 的研究上。概括來說，可採取的調查方法有兩種基本策略。

其一，提取軟體所有版本，分別計算每次版本的屬性，並進行比較，及間接的(外部的)測量軟體的演進。例如計算軟體複雜度、缺陷密度與可維護性等指標，進行軟體質量的評估。其二，此方法著重於版本間的具體差異，利用 CVS 版本控管系統或其他工具提供差異資料，進行直接的(內部的)軟體演進分析。例如進行實體程式碼的文件(file)、類別(class)、函數(function)變更的調查。

MSR 評估的指標也將資訊檢索(information retrieval，IR)領域的 precision 與 recall 等指標應用於此。另一種評估方法是採取信息理論方法(information theoretic approach)這被使用於預測軟體文件的變化與錯誤。而 MSR 的研究方法[2]概括分成四

層，如下圖 1，由下而上分別為(一)由底層直接存取資料來源，及程式碼文件、差異紀錄、元素數據等，稱之來源資料層(information sources)。(二)確保資料在同一水平線上來做分析所採用的MSR方法進行評估的分析框架，稱之展現層(representation)。(三)為了便於釐清目的且進行客觀性的調查，將其目的轉換成先前定義的兩類問題，Market-Basket Question（MBQ）與 Prevalence Questions（PQ），稱之目的層(purpose)。(四)藉由分析軟體屬性變化或版本實際差異所得出軟體演進的結論，稱之軟體演化層(software evolution)。

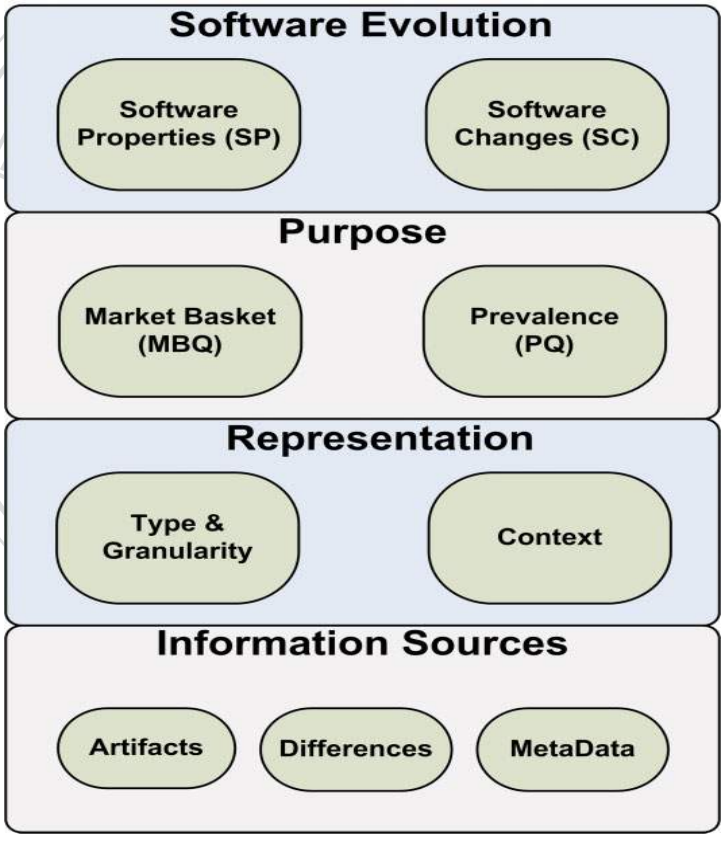

圖 1:MSR 方法分類分層[2]

MSR 的最終目標則是為了觀察出軟體系統的演進，發現軟體開發流程中的訊息、關係與趨勢，有助於軟體品質的評估，以利軟體開發持續進步；並且研究 MSR 分析工具與指標方法，使得能夠更準確的反應出軟體演進的過程。

2.3MSR 相關分析主題

MSR[2]所涵蓋的分析主題甚多，例如 CVS log 是一種最常被用來分析的資料種類，因為此類資料最容易取得，也可反應出軟體的真實情況，便於分析出軟體 bug 的增量，因此稱之此類的分析領域為 metadata analysis。或者透過分析靜態的原始碼(static source code analysis)，來了解軟體的歷史演進與個人版本的變化，進而比較不同版本，適用於發現系統漏洞與修復 bug。之後可進一步找出更好的方法去察覺程式版本的修改與差異(source code differencing and analysis)，例如透過語法或語義去識別程式碼的改變。也可以以定量方式進行整體軟體評估(software metrics)，衡量軟體各個方面，包括系統規模、工作量、成本、功能、質量、複雜性、效率、可靠度與系統可維護性等。

另一方面，則針對程式碼相似文本，結構和語義成分做分析，稱之 source code clone-detection methods。或者利用信息檢索(information-retrieval methods，IR)來進行分類或分群文本，應用於許多軟體工程的問題，例如問題可回溯性、程式理解分析以及軟體重複使用等問題。最後，透過機器學習的概念來進行分析(classification with supervised learning)，使用監督式學習讓機器不斷的整合資訊與預測結果，即利用訓練與測試資料兩種資料交互使用來形成分類與預測模型。而近幾年興起的社群網絡也可應用於 MSR 分析，利用社會與行為科學理論，評估或推論無形的社群網絡關係(social network analysis)，找出在軟體開發過程中，開發人員的角色與貢獻，並分析出軟體開發的過程中各種行為或是人員之間的關聯性，最後，再使用互動式的視覺化數據(data visualization)，可以具體的表示軟體維護與演進，增強使用者的認知。

2.4 資料集介紹

MSR 官方於 2014 年釋出 GitHub 平台上使用者操作與修改之紀錄，擷取原始的

GHTorrent 資料集[4]，其內容包含了前十大最常被加入最愛的專案所使用的程式語言，以確保資料集中能符合目前最流行的程式語言。整個資料集中包含了 90 個專案項目與複製專案(fork)相關紀錄。而資料集分為兩種資料庫格式，其一為 MySQL，另一種為 MongoDB，MongoDB 所儲存格式為 JSON(javascript object notation)。

資料來源部分除了直接從官方網站[5]上取得外，亦可使用呼叫 API 方式，即查詢 GitHub 上 API 的結果也可取得相同格式的資料。MongoDB 資料集還原後，有效資料表有 15 個，其資料表名稱、對應筆數與所涵蓋的時間區間等資訊彙整於表 1。

表 1:資料集表單說明、筆數與資料區間

資料表名稱	說明	資料筆數	資料區間
users	使用者	496,519	2007/10 - 2013/12
watchers	關注者	295,954	2007/10 - 2013/12
followers	跟隨者	1,596,888	2007/10 - 2013/12
org_members	組織成員	120,798	2007/10 - 2013/12
repo_collaborators	協同成員	1,941	2007/10 - 2013/12
repos	軟體庫	108,710	2008/02 - 2013/10
repo_labels	軟體庫標籤	1,206	2008/02 - 2013/10
forks	複製專案事件	107,984	2008/02 - 2013/10
commits	提交事件	601,080	2008/02 - 2013/10
commit_comments	提交事件留言	60,845	2008/02 - 2013/10
pull_requests	提交合併要求	79,359	2010/08 - 2013/10
pull_request_comments	提交合併要求留言	93,198	2010/08 - 2013/10
issues	問題	126,308	2010/08 - 2013/10
issue_events	問題提交事件	618,408	2010/08 - 2013/10
issue_comments	問題提交事件留言	583,794	2010/08 - 2013/10

而本研究使用 MSR 官方 2014 年所釋出的 MongoDB 資料集當作研究資料來源，資料期間涵蓋 2007/10/19 至 2013/10/10，且此資料集含有後續資料錯誤更正處理，至今已釋放四個更新版本，最後更新日為 2013/12/13。

[4] http://2014.msrconf.org/challenge.php
[5] http://2015.msrconf.org/

表 2:資料集更新時程與修正內容

版本	釋出日期	修正內容
1.3	2013/12/13	更正遺失的專案成員資料。
1.2	2013/10/22	更正 commit_comments 表單中的 user_id。
1.1	2013/10/09	更正部分專案遺失的 commits 與 comments 資料。
1	2013/09/28	

2.4.1 Table Schema

下圖(圖 2)顯示的是 MSR 於 2014 年所提供資料集當中，也就是本研究所使用的資料及當中，資料表的綱目(schema)。此綱目主要用途為了解各資料表之間的關聯與欄位定義，已利後續使用資料庫管理工具或撰寫程式取得欲分析之欄位。

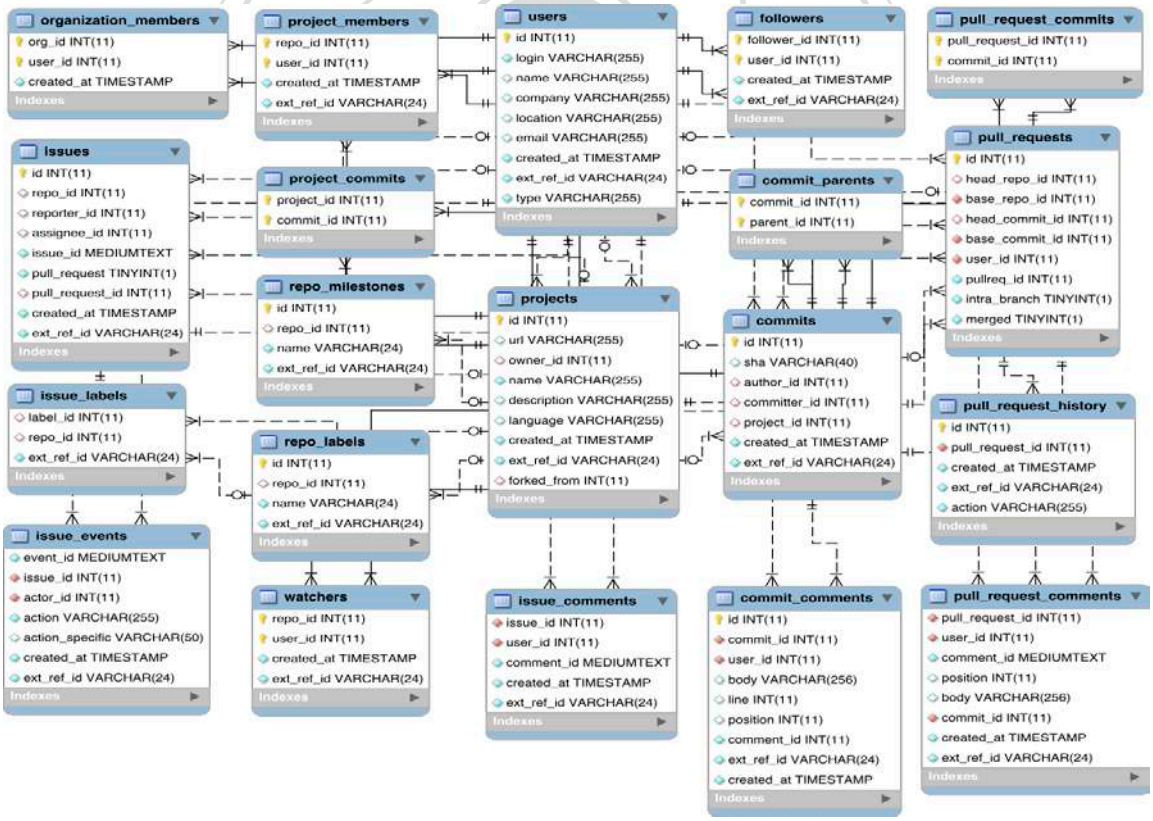

圖 2:MSR 2014 資料集的資料表綱目 [6]

[6] http://ghtorrent.org/relational.html

2.4.2 GitHub 相關名詞說明

以下針對 GitHub 上重要之名詞與操作事件進行說明,以下名詞亦使用於本研究之其他章節。organization members 表示此用戶是屬於哪一個組織(或公司)。repo collaborators 表示此用戶是屬於哪一個專案的協同開發成員，但不一定為同一組織。值得一提的是 follow 事件與 watch 事件，前者為追蹤另一用戶之動態，後者為關注某一專案之動態，關注後將會收到有關專案更新之訊息通知。fork 事件則是指用戶複製他人專案(軟體庫)至本地端，且納入本身專案(軟體庫)，進行協同開發。之後透過 commit 事件提交程式源碼至本身或組織的專案中。若用戶修改他人專案後，可利用 pull request 事件提出要求加入(或合併)至專案版本中，而專案擁有者可決定是否合併此版本至專案中，以發佈成最新版本。最後，GitHub 針對軟體庫相關的問題，區分出數種觸發事件，例如用戶可訂閱或退訂問題的通知訊息，或者當問題中提到另外一位用戶,可使用@User 方式來表示此問題提到此用戶,另外有引用或分配問題、關閉或重新啟動問題等多種觸發事件可供 GitHub 用戶使用。

2.5 資料樣本結構分析

2.5.1 事件總數

在進行研究之前，本論文先針對 MSR 資料集做基本統計分析，以了解資料細節與全貌。也可透過統計來初步觀察目前資料所能提供分析之訊息。承接上一章節所介紹 GitHub 常見之名詞，先針對 GitHub 平台上常用的五種操作事件，統計出資料集中這五種事件的分佈情況，如下圖 3。數量由多至寡，事件依序為 follow、commit、watch、fork 與 pull request，且 follow 事件超過次多的 commit 事件 1 倍以上，而資料集中數量最少的 pull request 事件與數量最多的 follow 差距約 20 倍之多，結果可明顯發現，目前 GitHub 平台上大多數使用者仍以追隨(follow)或關注(watch)等被動式的操作為主，藉此比較 follow 與 watch 兩種事件，可初步得知目前 GitHub 會吸引

大家使用的是”人”，而非”專案”，簡單來說，使用者會以追隨(follow)某人為主要操作行為，並非關注(watch)專案。

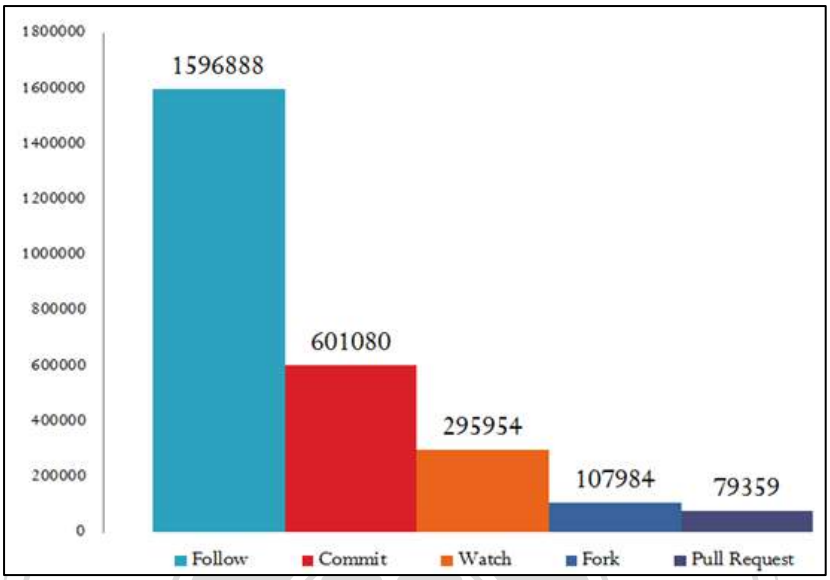

圖 3:GitHub 上五種事件次數總計

2.5.2 專案中最常使用的程式語言

本小節計算 MSR 資料集中專案所使用的程式語言來進行排名(圖 4)，並與 GitHub 官方資料所統計的程式語言排名(表 3)做比較。結果發現排名與官方整體資料差異不大，表示此資料集所呈現的實際情況與 GitHub 平台上的現有資料極為接近。因此，使用此資料集作為本研究基礎更能反映 GitHub 平台上的實際情況。

TIOBE 是一個程式語言的流行與趨勢的指標(圖 5)，每月會進行調查而產生程式語言排行榜，開發人員可以依據此排行榜來檢視自己所學習的技術是否有跟上潮流與趨勢。能夠被列在 TIOBE 的排行榜中，主要必須符合以下三點(一)必須在 Wikipedia 有專屬的詞彙說明，且明確指出是一門程式語言，且必須符合圖靈完備(turing-computable)[7]。(二)語言的排名根據在各大網站的搜尋引擎所搜尋出結果數量

[7] http://planetmath.org/turingcomputable

的平均值當作排序指標[8]。(三)平均值(rating)連續三個月高於 0.7 則會列入主流語言排行榜。TIOBE 排名計算中最主要的指標為主流搜尋引擎的搜尋數量。而其中如何定義主流的搜尋引擎，TIOBE 則是由 Alexa.com 網站[9]上的排名所決定。而利用八個主流搜尋網站(Google、Blogger、Wikipedia、YouTuBe、Baidu、Yahoo!、Bing、Amazon)的結果數量當作 TIOBE 排名計算的依據。而本研究利用 MSR 資料集或 GitHub 官方資料所觀察出的程式語言排行，與 TIOBE 的程式語言排行榜進行比較，觀察發現 TIOBE 與其他兩者差異較大，爾後將 MSR、GitHub 官方與 TIOBE 三者的程式語言排名差異進行量化分析，本研究利用斯皮爾曼等級相關係數[10]進行三者之間相關係數的比較，結果發現 MSR 與 GitHub 官方所呈現的程式語言排名相關程度較高，相關係數約為 0.76；而 MSR 與 TIOBE 的相關係數則偏低，相關係數約為 0.48。表示目前 GitHub 平台上各專案所使用的程式語言，或是專案的協同開發人員所使用的程式技術，未能反映出目前全世界多數程式開發人員所使用的主流技術。

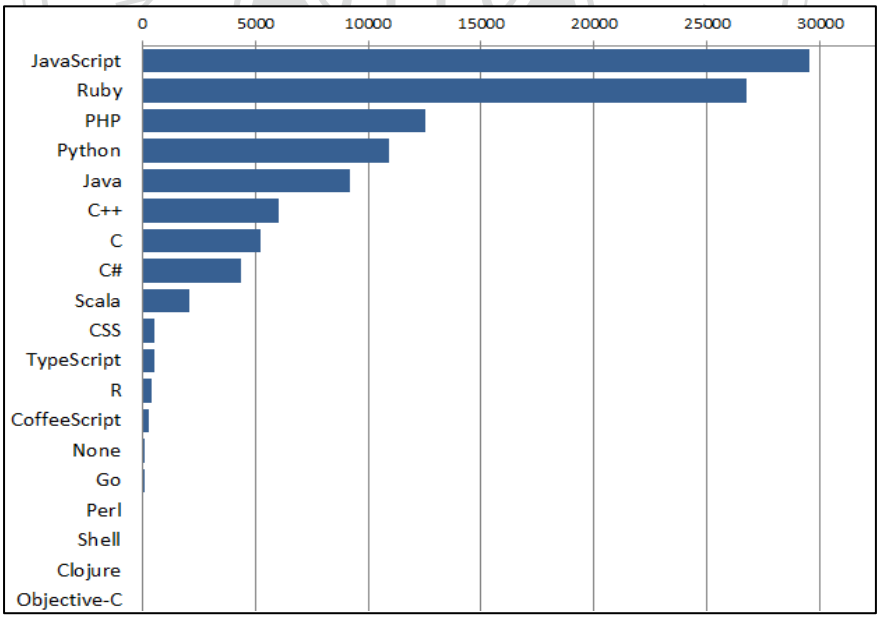

圖 4:MSR 資料集程式語言排名

[8] http://www.haodaima.net/art/2082645
[9] http://en.wikipedia.org/wiki/Alexa_Internet
[10] https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

表 3:2014 年 GitHub 專案程式語言排名 [11]

Language	Rank			# New Repos Created		
	2014	2013	2012	2014	2013	2012
JavaScript	1	1	2	383,185	320,534	277,875
Java	2	3	3	283,354	185,530	240,992
Ruby	3	2	1	259,268	228,145	310,281
C	4	7	4	178,891	79,223	203,992
CSS	5	12	25	175,573	18,869	3,791
PHP	6	4	6	175,476	139,591	157,185
Python	7	5	5	151,669	126,027	165,655
C++	8	6	7	78,878	104,499	88,615
Objective-C	9	8	11	60,579	40,072	36,539
C#	10	10	10	59,472	34,992	39,486
Shell	11	9	8	48,388	35,204	68,720
R	12	23	26	25,229	3,790	3,216

Jun 2015	Jun 2014	Change	Programming Language	Ratings	Change
1	2	^	Java	17.822%	+1.71%
2	1	v	C	16.788%	+0.60%
3	4	^	C++	7.756%	+1.33%
4	5	^	C#	5.056%	+1.11%
5	3	v	Objective-C	4.339%	-6.60%
6	8	^	Python	3.999%	+1.29%
7	10	^	Visual Basic .NET	3.168%	+1.25%
8	7	v	PHP	2.868%	+0.02%
9	9		JavaScript	2.295%	+0.30%
10	17	^^	Delphi/Object Pascal	1.869%	+1.04%
11	-	^^	Visual Basic	1.839%	+1.84%
12	12		Perl	1.759%	+0.28%
13	23	^^	R	1.524%	+0.85%
14	-	^^	Swift	1.440%	+1.44%
15	19	^^	MATLAB	1.436%	+0.66%

圖 5:TIOBE 程式語言排名(於 2015 年 06 月 28 日取得之結果)[12]

[11] http://adambard.com/blog/top-GitHub-languages-2014/
[12] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

2.5.3 最常被關注的專案

GitHub 平台上使用者可以針對自己有興趣的專案進行關注(watch)動作，因此，關注(watch)這項操作事件便能表示使用者跟隨自身興趣或潮流趨勢的一種指標。而本小節則針對 MSR 資料集中使用者所關注(watch)的專案作為觀察對象，並比較在 MSR 資料集中，與 GitHub 官方資料或 TIOBE 排名，是否具有相似或一致性，或是否同樣能表示為程式設計的趨勢，進而觀察出此 MSR 資料集是否接近現實情況，並驗證此資料集的可信度。

表 4:GitHub 上最常關注之專案所使用語言列表

排名	專案名稱	使用語言	關注次數
1	Node	JavaScript	24,559
2	jquery	JavaScript	23,692
3	html5-boilerplate	CSS	22,292
4	Rails	Ruby	19,587
5	d3	JavaScript	18,365
6	impress.js	JavaScript	17,513
7	Font-Awesome	CSS	16,972
8	homebrew	Ruby	13,870
9	chosen	JavaScript	13,489
10	foundation	JavaScript	13,358
11	three.js	JavaScript	12,733
12	Jekyll	Ruby	12,359
13	gitlabhq	Ruby	10,244
14	devise	Ruby	9,167
15	diaspora	Ruby	8,589
16	phantomjs	C++	7,723
17	django	Python	7,404
18	Flask	Python	7,324
19	Redis	C	7,154
20	symfony	PHP	7,103

如上表 4，將使用者最常關注的專案進行排名，並將前二十名的專案與其使用的程式語言進行整理彙整成表格。並承接上一小節，利用斯皮爾曼等級相關係數計

算出 MSR、GitHub 官方與 TIOBE 三者之間程式語言排名的相關程度，結果發現 MSR 與 GitHub 官方呈現正相關，相關係數為 0.27；而 MSR 與 TIOBE 呈現微弱相關，相關係數為-1.5。

2.5.4 統計發現

透過 MSR 資料集的基本統計後，得出四點發現。其一，資料集中顯示關注(watch)事件遠超過於其他操作事件。GitHub 理想目標為開放源碼，鼓勵使用者能夠透過協同寫作方式[17]，多方參與軟體專案的開發，並且貢獻原始碼。但依照資料集中顯示的結果，呈現多數使用者仍然以觀看專案作為主要用途，僅少數人會進行專案的參與，即資料集中呈現的提交需求(pull request)為數量最少的操作事件。所以目前 GitHub 上的使用者多數進行單向的互動，如關注(watch)、跟隨(follow)、留言(issue)等，反觀貢獻者或協同開發者較為少數。

其二，因近年來大數據的發展，讓利於統計分析與運算的 MATLAB 與 R 等程式語言更為主流，在 TIOBE 的排名中也可看見此兩種語言近幾年主流的程度已漸漸提升，圖 5 顯示 2015 年排名已上升至 13 名與 15 名。但在 MSR 資料集卻未看見使用 MATLAB、R 或是其他利於統計分析的程式語言相關專案有大幅的成長。其三，本研究透過資料集中程式語言的排名(圖 4)與最多人關注的專案(表 4)進行比較與觀察。觀察出 GitHub 平台上最常使用的程式語言，可能並非大家所感興趣的語言。

圖 4 呈現 MSR 資料集中有不少使用 Java 程式語言的專案，專案數量約為八千至九千，但反觀表 4，在針對使用者關注的專案中，卻顯示目前較少數人關注以 Java 語言為主的專案。如此可推論目前 Java 語言非目前大眾所感興趣的主流程式語言。反觀 JavaScript 與 Ruby 兩種語言不僅是專案中最常使用之語言，同時也為最多人關注的語言，兩圖表交叉比對下，更能反映目前此兩種程式為目前主流學習語言，也與大眾理解中的相符合。

最後，從 MSR 資料集或 GitHub 中觀察出近似於長尾效應現象[13]，傳統長尾效應表示為百分之八十的使用者在關注著百分之二十的專案，剩餘百分之二十的使用者關注其他百分之八十的專案。表示熱門的專案具有吸引力並取得大家關注，但不受使用者關注或不感興趣的專案卻佔 GitHub 中大多數。但實際從資料集中觀察，前百分之二十的專案，僅獲得 48%的關注人數，反之剩餘百分之八十的專案，反而獲得 52%的關注人數。這表示以目前資料集來看 GitHub 上的使用者較不容易產生跟隨熱門專案之情形，即使較為冷門的專案也會有不少使用者進行關注。

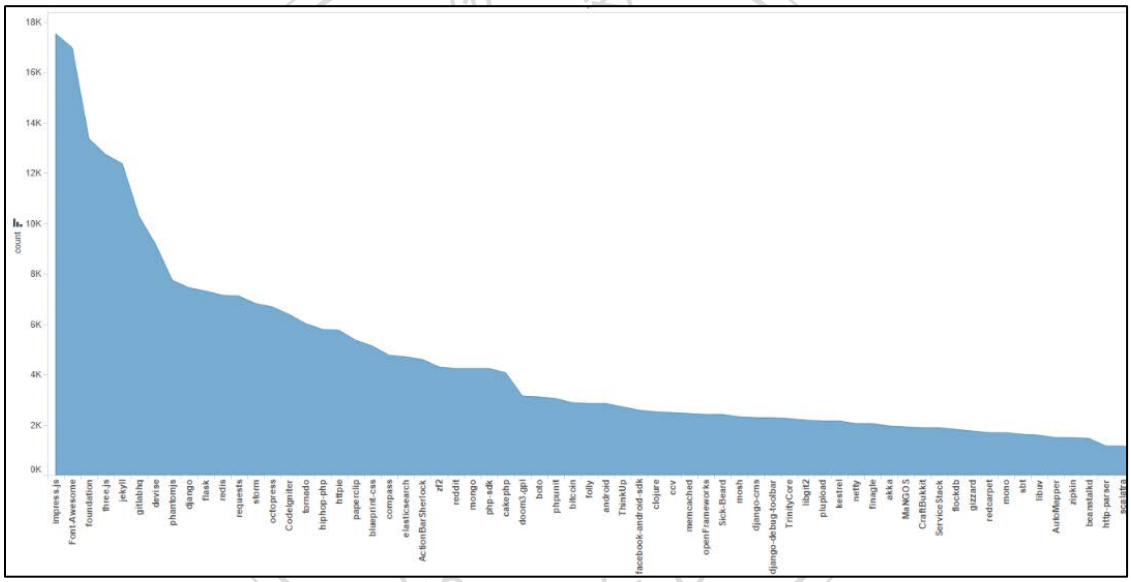

圖 6:GitHub 上的長尾效應現象

[13] http://en.wikipedia.org/wiki/Long_tail

第三章、協同寫作與社群網絡

3.1 資料前置處理

進行社群網絡分析之前，先行針對資料集做資料過濾與清理，本研究採用的技術與工具為 Python、MySQL 與 MongoDB，並使用 Robomongo 與 Rockmongo 等兩種資料庫管理工具。資料集部分至 MSR 官方網站上取得 2014 年所釋出的 MongoDB 版本，壓縮檔約為 2.7GB。安裝 MongoDB 後，利用 Robomongo 與 Rockmongo 兩種管理工具進行還原動作，還原後資料庫大小約為 34GB。隨後利用 Python 程式語言連結 MongoDB 來讀取所需欄位與資料，並同時過濾資料異常值，例如 N/A 值或是中文亂碼等，並寫入 CSV 檔。由於 MongoDB 中所儲存的格式為 JSON 格式，較難進行表單的合併(join)與群組(groupby)，因此將 MongoDB 所匯出的 CSV 檔，再匯入至 MySQL 進行 SQL 語法操作，以便進行資料的處理。

本研究針對社群網絡分析所採用的分析工具為 Gephi[14]視覺化工具，此工具必須讀取網絡圖中相對應的節點(node)與連結(edge)資料。因此，在社群網絡分析部分也利用 MySQL 工具轉換成 Gephi 工具所需之節點與連結兩種檔案。研究過程中由於資料集筆數過於龐大，在資料過濾中先行排除未活動(零活動)之使用者，或未活動之專案。若使用者尚未觸發任何事件或觸發事件較少者也將不列入分析範圍;專案資料處理方式亦同，排除未有使用者關注，或是未觸發提交事件的專案也將不列入分析。將未活動之使用者或專案視為離群值，於予排除，以提升資料有效性與正確性，避免造成離群或異質資料影響資料整體分析。

3.1.1 資料定義

由於資料集筆數過於龐大，因此，在研究前將事先排除未具有活性的使用者或專案，

[14] http://gephi.github.io/

本章節將進一步說明本研究如何定義活性，與過濾資料的條件與範圍。

針對使用者，原資料集中筆數為 496,492，其中帳戶型態為組織有 31,318 筆，一般使用者有 465,174 筆。刪除資料異常 1,321 筆後剩餘 495,171 筆。將資料集中以下四個欄位設定為定義使用者活性之標準，(一) followers，此人被追隨之數量。(二) following，此人追隨他人之數量。(三) public repos，此人公開專案之數量。(四) public gists，此人使用 Gist 服務[15]之數量。並將此四個欄位數值加總，計算出活性值。欄位數值加總後，活性值為 0 的共有 27,738 筆，約佔資料集的百分之 6 (27738 / 495171 = 0.056)。實際觀察資料分布後，發現資料集中多數為低活性之使用者，為了追求研究數據之合理性下，取最大活性值的百分之一作為資料刪除標準，因此，保留活性值大於 220 以上資料，共 3,995 筆，其中使用者型態為組織有 66 筆，一般使用者有 3,929 筆，計算方式如下:

$$\text{Max(Active values)} * 0.01$$

$$21975 * 0.01 = 219.75 \fallingdotseq 220$$

針對專案，原資料集中筆數為 108,541，其中帳戶型態為組織有 3,844 筆，一般使用者有 10,496 筆。刪除資料異常 12 筆後剩餘 108,529 筆。將資料集中以下三個欄位為定義專案活性之標準，(一) watchers，此專案被關注之數量。(二) forks，此專案被複製之數量。(三) open issues，此專案留言之數量。將此三個欄位數值加總，計算出活性值。欄位數值加總後，活性值為 0 的共有 67,453 筆，約佔資料集的百分之 62 (67453 / 108529 = 0.62)。而後發現資料集中多數為低活性之專案，為求數據合理性，取最大活性值的百分之一作為資料刪除標準，因此保留活性值大於 300 以上資料，共 100 筆，其中使用者型態為組織有 60 筆，一般使用者有 40 筆，計算方式如下:

$$\text{Max(Active values)} * 0.01$$

$$29929 * 0.01 = 299.29 \fallingdotseq 300$$

[15] https://help.github.com/articles/about-gists/

3.1.2 資料前置處理流程

針對資料集的前置處理，其流程分為以下五個步驟:

資料剖析:(一)首先將資料集還原至 MongoDB，並對照本論文 2.4.1 章節的資料庫關聯圖進行欄位比對，針對 JSON 格式確認所有欄位的定義，了解其資料結構。利用 Python 程式語言取得欲分析的欄位，並將其匯出成 CSV 格式，或轉至 MySQL 資料庫，建置資料關聯性，以利處理後續資料清理之步驟。

資料亂碼處理:(二)資料集還原後，有部分資料為亂碼或異常值。由於目前 GitHub 平台上有來自其他國籍的使用者帳戶，因此資料還原後不同語言會有文字編碼問題而產生亂碼。此情形大多出現在使用者資訊，如居住地址、公司名稱等。亦或是需要使用者自行輸入之欄位較常出現不符欄位定義之異常值，因此，在此步驟則針對此類資料進行字元取代或刪除等動作。

資料空值與欄位錯位處理:(三)資料集中空值部分有許多不同的顯示方式，如 N/A、Null 等。在此步驟則統一將空值都表示為”None”，以利後續當作過濾條件，或便於資料分析使用。此外，由於資料還原後為多階層之 JSON 格式，因此轉至 MySQL 資料庫時會有欄位對應錯誤之問題，一併於此步驟進行修正或刪除。

針對使用者表單進行加總計算與資料篩選:(四)此步驟針對本論文 3.1.1 章節所定義的使用者活性，從 user table 取得相關欄位，取得 followers、following、public repos、public gists 等四個欄位後，並轉至 MySQL 資料庫進行加總與計算，最後篩選出具有活性的使用者資訊，以利後續研究分析之有效性與正確性。

針對專案表單進行加總計算與資料篩選:(五)同步驟四，根據專案活性的定義，從 repo table 取得 watchers、forks、open issues 等欄位，並轉至 MySQL 資料庫進行加總與計算，最後篩選出具有活性的專案資訊，以利後續研究分析之有效性與正確性。

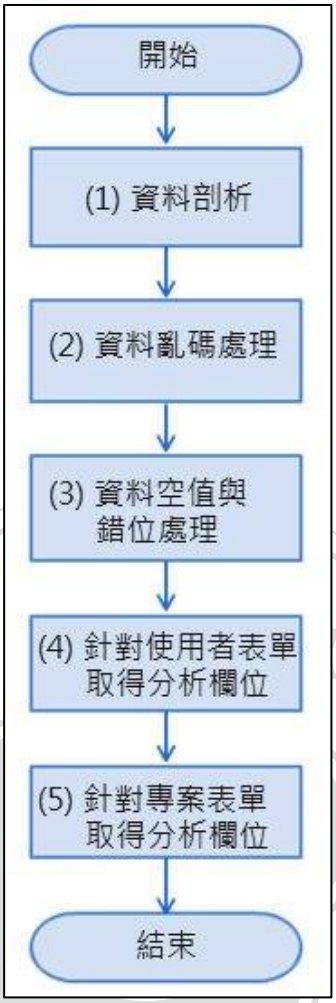

圖 7:資料前置處理流程

3.2 社群網絡觀察

3.2.1 以關注事件(watch)觀察網絡現象

社群網絡(social network)一詞[16]是 1954 年由 J. A. Barnes 首先使用（於 Human Relations 文中的 Class and Committees in a Norwegian Island Parish 章節內）。社群網絡分析是用節點與連結的概念來檢視之間的相關聯性，節點表示為網絡中的個人參與者，為一單獨個體，連結則表示為這些單獨個體之間的關係，節點之間能夠同時具有多種連結。目前部分學術研究已經顯示，社群網絡在很多層面進行運作，並扮演

[16] http://en.wikipedia.org/wiki/Social_network

著關鍵作用，從組織運行、分析決策到解決問題，並在某種程度上可預測未來發展趨勢。

社群網絡分析以廣義而言能夠橫跨多方領域，如社會學、媒體傳播、生物學、物理學等，起初社群網絡分析大多發展於社會學系領域，從個人的社會脈絡，到組織內部資訊傳遞，甚至大到國家之間的網絡分析；社群網絡亦可應用於生物醫療領域上，例如基因之間的關聯或流行性病毒的傳染等。以往在媒體傳播領域中分析社群網絡，是以傳統的報章雜誌去觀察媒體間的網絡關係，然而隨著網路的發展，資訊量的遽增，傳播媒體的方式改變。因此，便興起了在電腦科學領域上的社群網絡，利用科學的角度去分析以網路為媒體的資料，並觀察其社群網絡關係。

本章節亦在此基礎下研究，其目的為觀察 GitHub 平台上使用者所關注的專案是否有相同屬性，存在什麼因素會影響他們關注某一類型的專案[7]，並更進一步了解專案之間的社群網絡關係，而網絡中的關聯為使用者是否關注某一專案。首先，建立單一模式網絡(one-mode network)，即網絡圖形中僅包含單一屬性節點。以專案當作節點(node)，並利用使用者是否同時關注某一專案當作網絡關係連結(edge)，並以是否連結到相同專案當做權重(weight)，最後利用社群網絡分析工具 Gephi 產生視覺化圖形，本研究在進行 Gephi 工具操作時，設定以 weight degree 表示節點大小，並以 modularity[24]當作分群的指標，隨後設定工具中 modularity 的 randomize 與 resolution[25]兩項參數，勾選 randomize 參數可以利用原有資料隨機產生另一個網絡，並與實際資料的網絡進行比較，直到所有節點都產生分群，然而 randomize 能夠讓每回合運行都找到最好的分群方式，並改善分群效果。而 resolution 參數會影響分群的數目與大小，若設置較低的 resolution 值，則會產生較多但規模較小的分群，反之，設置較高的 resolution 值，則產生較少但規模較大的分群，而本研究目前設置為 Gephi 工具的 resolution 預設值 1.0，可依資料特性看是否需要觀察較為細節或概括的網絡，而調整 resolution 數值大小。隨後即產生以專案為主體的社群網絡圖，並觀察出不同

類型的專案間所存在的關聯，如圖 8 所示。

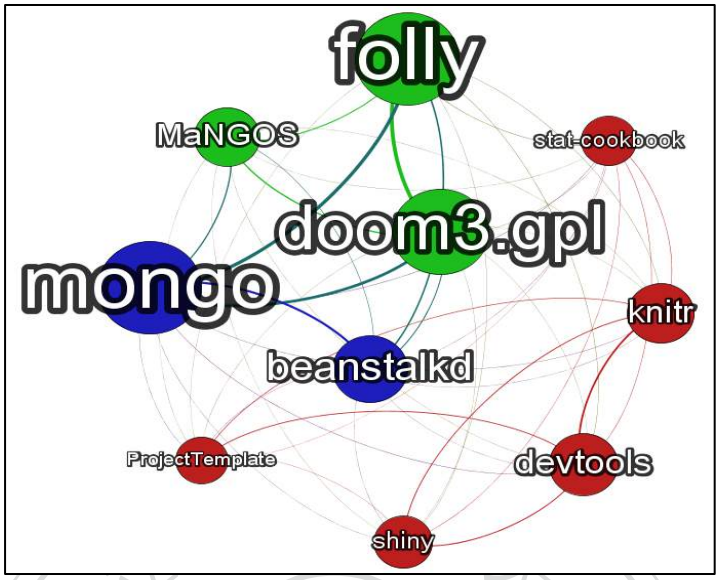

圖 8:關注事件與專案之社群網絡圖

圖形結果顯示 doom3.gpl、folly、MaNGOS 具有較高的關聯程度。而後將圖形關聯轉換成以下表格，並列出專案的擁有者與使用語言。表 5 以關聯性強的專案顯示為同一群組號碼，結果可發現具有高度關聯性的專案通常是相同類型的程式語言。以下表格顯示使用者在關注專案時，通常以自身領域有相關的技術，鮮少會關注不同技術的專案，如 doom3.gpl、folly、MaNGOS 三個專案之間具有關聯性，且都使用 C++語言，而 stat-cookbook、ProjectTemplate、devtools、knitr 專案具有關聯性，且都使用 R 語言等。

表 5:具有高度關聯的專案

分群號碼	專案名稱	擁有者	使用語言
1	doom3.gpl	TTimo	C++
1	folly	facebook	C++
1	MaNGOS	mangos	C++
2	mongo	mongodb	C++
2	beanstalkd	kr	C
3	stat-cookbook	mavam	R
3	ProjectTemplate	johnmyleswhite	R
3	devtools	hadley	R

3	knitr	yihui	R
3	shiny	rstudio	JavaScript

3.2.2 以合併要求事件(pull request)觀察網絡現象

此章節主要探討的主題有以下三個方向(一) 觀察 GitHub 上關注(watch)與合併要求(pull request)此兩種事件與專案之間所存在的關聯。(二) 觀察是否有使用者同時提交合併要求(pull request)好幾個專案[10]，表示此使用者相較於其他人具有較高的提交程式碼之能力。(三) 觀察是否有專案尚未受到關注(watch)，但已經有多人提交合併要求(即貢獻源碼)，換言之，此專案已有多人進行協同寫作，但尚未受到關注，表示此專案是具有潛力的專案。

首先，建立雙模式網絡(two-mode network)，網絡圖形中節點(node)包含使用者和專案兩種型態節點，並以顏色區分專案(藍色)、觸發關注事件使用者(紅色)，或觸發合併要求事件使用者(綠色)，因此，網絡圖形呈現三種顏色節點。當使用者關注或提交某一專案當作連結(edge)，並分別針對關注事件與合併要求事件設定不同權重(weight)，關注事件設定較小權重，反之合併要求事件設定較高權重，如此在建立網絡圖形時就能利用權重來區分這兩種節點對於專案的影響程度。

圖 9 為關注與提交事件與專案之網絡圖形，紅色節點表示為關注事件，綠色節點表示為合併要求事件，藍色節點為專案。資料集中關注節點占 72.59%，提交節點占 27.1%，專案節點占 0.31%。然而因為關注事件權重較小所以距離專案的節點較遠，反之提交事件權重較大，因此，會在專案節點周圍形成一圈綠色節點。由下圖可看出目前 GitHub 上仍以關注事件為主要操作行為，進行合併要求事件而產生貢獻的使用者反而是較為少數。

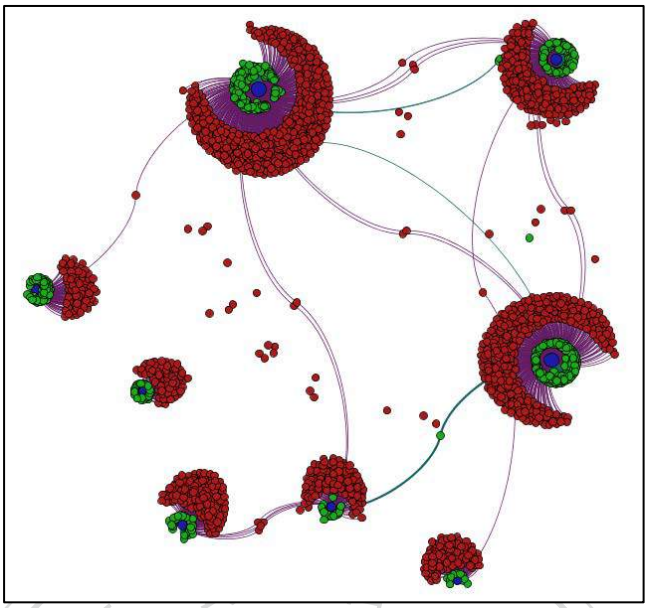

圖 9:關注與提交事件與專案關聯

下圖 10 為表示有一位使用者同時提交了兩個以上的專案，資料顯示為使用者 saschpe 同時提交給 libgit2 與 django 兩個專案，表示此使用者相較於其他人具有較高的提交程式碼之能力。但從目前的資料集來看，此類型的使用者目前僅有四位使用者，分別為 saschpe、kaz29、openam 與 CodeBlock，表示目前 GitHub 上同時對多個專案進行提交的貢獻者占為少數。

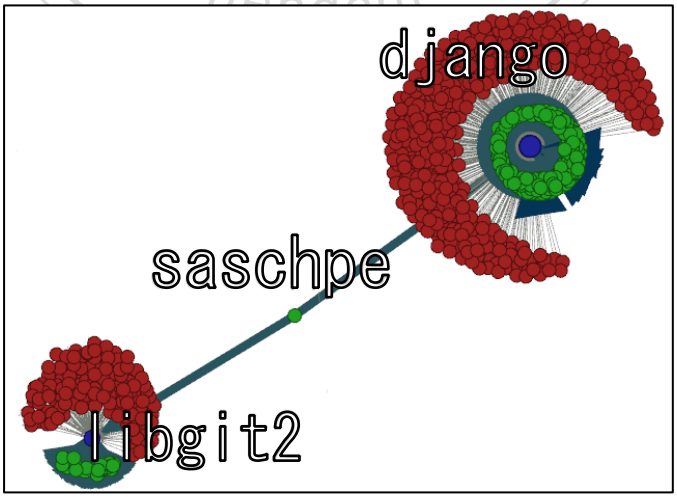

圖 10:具有較高提交程式碼能力之使用者

圖 11 顯示出目前專案的關注事件仍遠超過合併要求事件。所以若此專案的合併要求事件相較於關注事件的數量來的接近或平均，則表示此專案目前已有較多人進行合併要求(即貢獻源碼)，但尚未受到關注，此類型的專案則列為較有潛力的專案。圖中表示右上角的 gitlabhq 專案為較熱門的專案，有不少合併要求事件，但關注事件仍為大宗，共有 577 筆關注事件與 139 筆合併要求事件，合併要求比例占 0.19，其他專案的情況雷同，大多數為關注事件遠超過於合併要求事件。但反觀左下角的 mono 專案，提交事件與關注事件數量較為接近，共有 92 筆關注事件與 81 筆合併要求事件，合併要求比例占 0.47，所以將此列為具有潛力之專案。

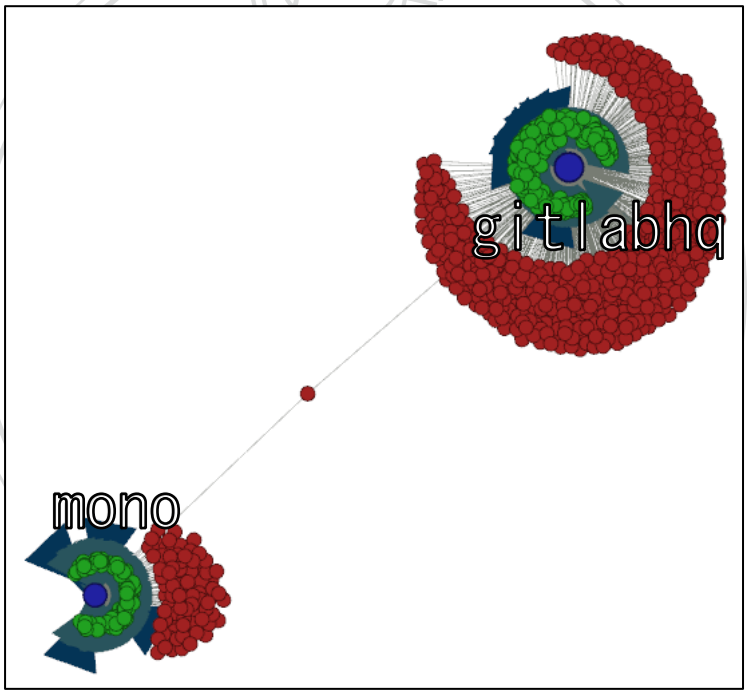

圖 11:具有潛力之專案

3.2.3 以程式語言與公司組織觀察網絡現象

本小節所研究的主題為觀察 GitHub 上各專案所使用的程式語言之間是否存在關聯性，透過此關聯性能夠觀察出哪些程式語言容易在協同寫作中同時出現，且開發人員同時參與了兩種以上不同的程式語言專案。本研究的方法以 GitHub 上協同寫作的

資料為基礎，即取得所有專案共同參與的開發人員，若兩個專案之間有相同的參與者則產生關連，最後透過專案所使用的程式語言屬性來產生社群網絡圖形。網絡圖形中任兩種程式語言有強度關聯，所表示的涵義為在 GitHub 上的開發人員會 A 語言，同時也會 B 語言的機率很高，下圖 12 呈現程式語言關聯性之網絡圖形。

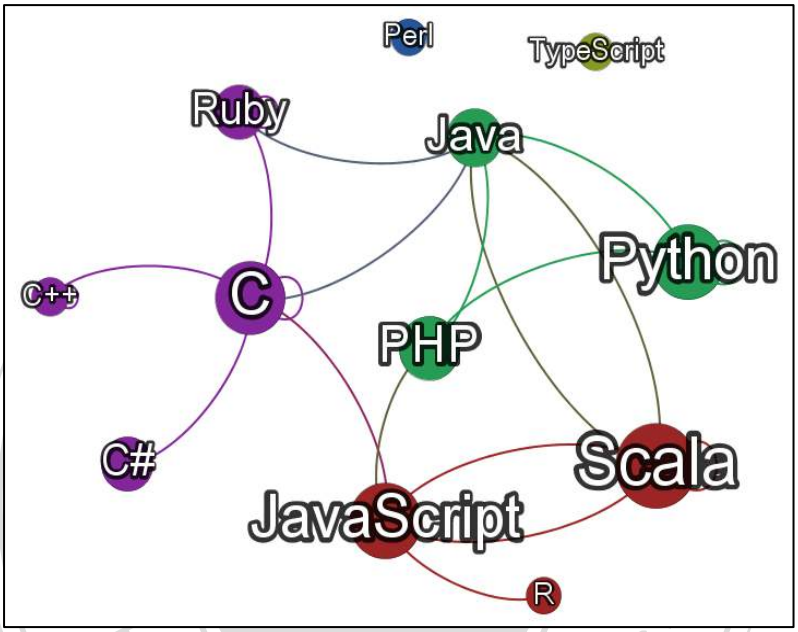

圖 12:程式語言關聯性

承接上述資料，進一步延伸相關議題，以公司組織的角度進行社群網絡觀察，目的為觀察出組織之間是否有相關聯性，作法同樣使用兩個專案之間是否有相同的參與者來產生關聯，最後透過專案所屬的公司組織來產生社群網絡圖形。因此，在網絡圖中任兩個組織有強度關連，則表示在 GitHub 上的開發人員同時會對這兩個組織進行貢獻的機率很高，下圖 13 呈現公司組織關聯性。

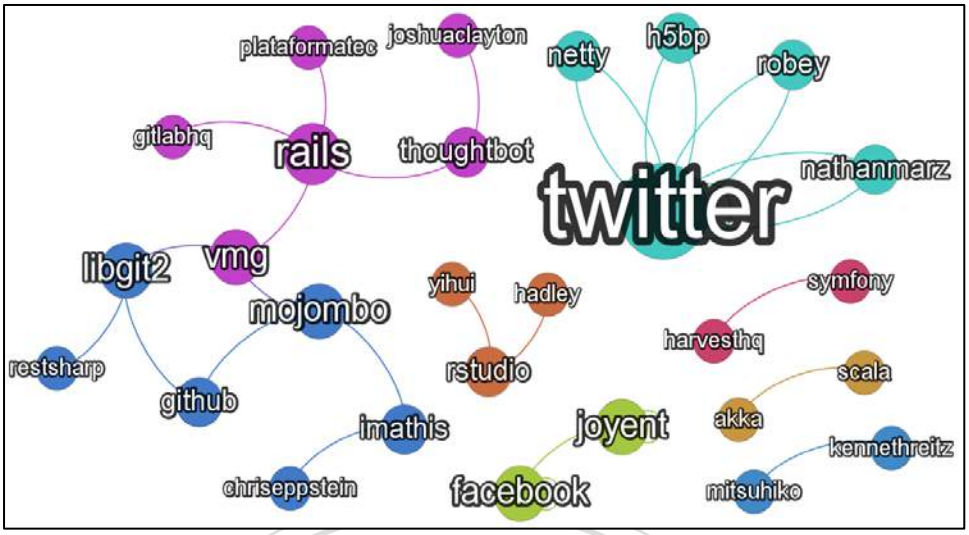

圖 13:公司組織關聯性

3.3 社群網絡指標分析

3.3.1 平均 Degree 與 Degree 分佈

本章節主要以使用者貢獻(pull request)與追隨(follow)等兩種角度分析，觀察此兩種事件所產生的兩種網絡。此兩種網絡的共通性即網絡圖形中的節點皆為使用者(user)，即以使用者角度而非專案角度去看整體的網絡，而節點間的連結則是以兩位使用者之間是否產生貢獻或追隨等觸發事件。

此小節使用目前社群網絡分析最常用的數種指標來分析此兩種網絡。上一章節我們僅透過初步的觀察來針對 GitHub 的整體社群網絡做初步的推論。而透過社群指標的分析，便能更進一步的去分析網絡中的種種細節，例如分析出網絡中每位使用者的貢獻與合作情況、或找出網絡中最具影響力的使用者等。以下我們先透過平均 degree 與 degree 的分佈來看這兩種觸發事件所反映的實際情況。

在追隨(follow)的原始資料中具有大量零活性(活性的定義請見 3.1.1)的使用者資料，所以在計算 degree 之前先進行資料篩選與過濾，先行去除零活性的使用者資料。去除零活性的使用者前資料共有 1,596,888 筆，資料篩選後僅留下 3,757 位使用者與 109,901 筆資料。

追隨(follow)的平均 degree 為 29.25，中位數為 14，標準差為 29.3±68.6，表示 GitHub 上一個人平均追隨 30 個人，因為在 GitHub 上使用者追隨某一使用者僅會有一次的關聯存在，即表示資料中不會顯示使用者多次追隨某一使用者，因此，此圖形是一種沒有權重的網路，其中節點間的關聯權重都為 1，表示權重都相同。而由下圖 14 可觀察出 degree 的分佈呈現 power law 現象，即只有少數使用者具有高度的 degree，大部分的使用者的 degree 大部分都介於 10 以下，表示在 GitHub 上的 follow 觸發事件呈現了一種大部分的使用者都共同追隨某一個使用者的現象，造成某幾位使用者擁有高度 degree，然而大部分的使用者並不會互相追隨。這也表示在 follow 的網絡下，各使用者間的連結性是不足的，屬於低互動的社群網絡。

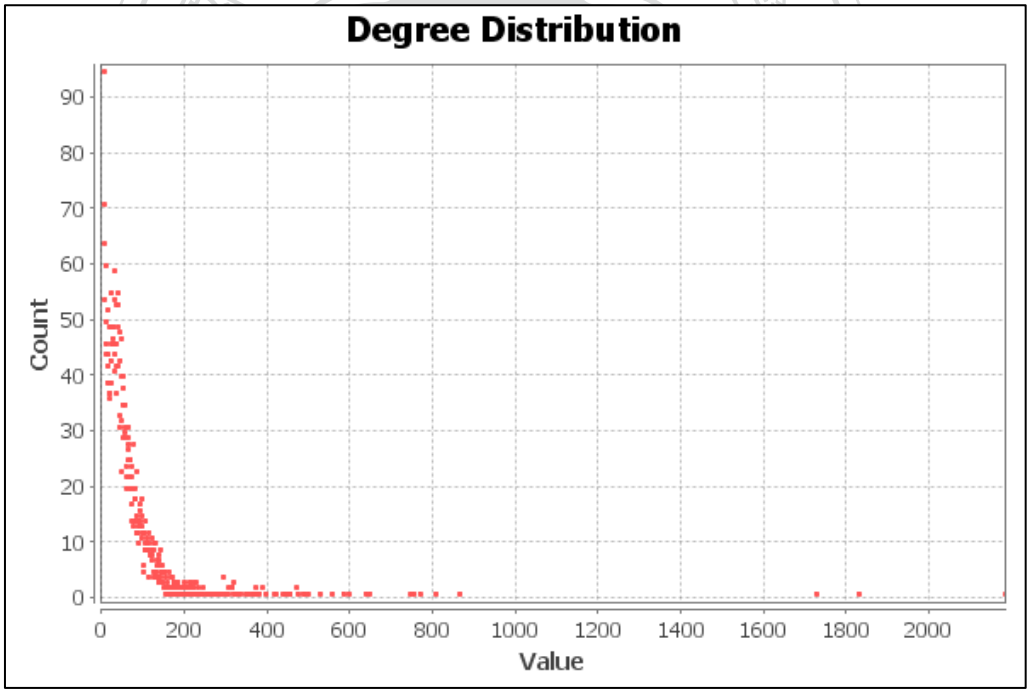

圖 14:Follower Degree Distribution

接下來我們針對使用者貢獻(pull request)的 degree 分佈來觀察，GitHub 上使用者貢獻所產生的真實情況。同樣的我們先過濾資料中零活性的使用者，留下 682 位使用者與 805 筆資料。使用者貢獻(pull request)的平均 degree 為 1.18，中位數為 1，

標準差為 1.2±0.5，因為可能使用者對某一使用者進行多次貢獻(pull request)，所以會產生出一個有權重的網絡圖形，此網絡則以貢獻給相同使用者的次數加總，作為關聯之間的權重，進而產生此網路圖形的 weight degree 為 6.03。由下圖 15 看出 degree 的分佈不同於 follow 事件，degree 的分布較為平均，分布於都介於 0 至 20 左右，只有一人的 degree 高於平均值很多，表示此人很多人貢獻給他或貢獻給很多人，在本研究中此人為貢獻給多人，即為 out degree。藉由此圖也觀察出目前 GitHub 上針對貢獻(pull request)觸發事件，大家的貢獻程度都較低，鮮少有多數人都貢獻於一人之情況。

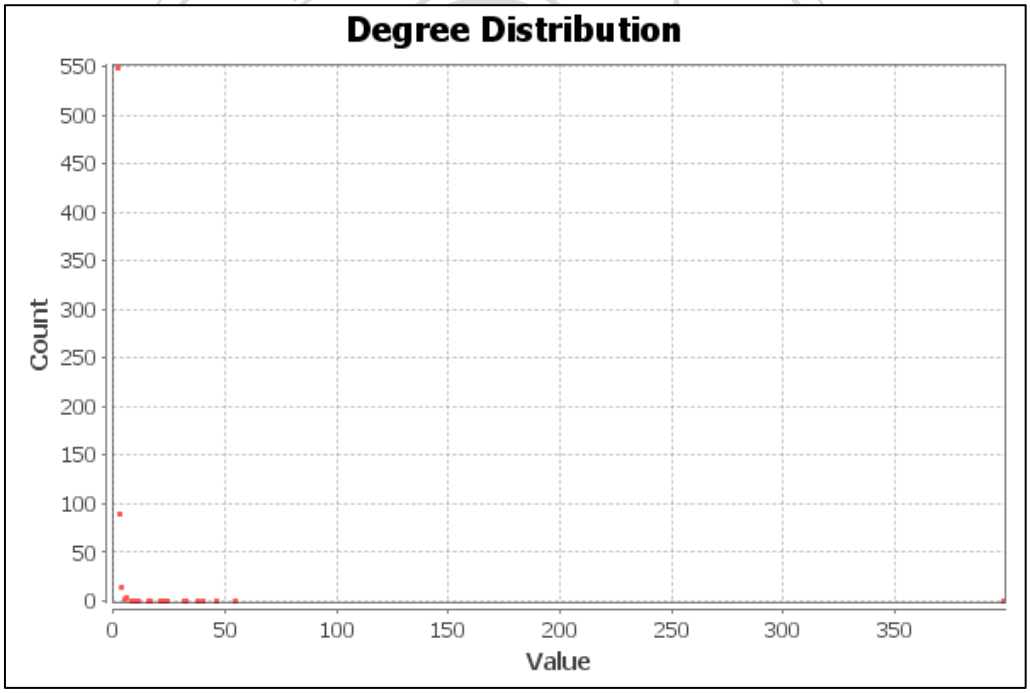

圖 15:Pull Request Degree Distribution

Degree centrality、Betweenness centrality、Closeness centrality[12]在社群網路分析三個重要的指標，能夠分析出網路中每個節點的中心性程度，即能夠表示每個節點的影響力與重要程度。

Degree centrality 的基本涵義是任一個節點連接到其他節點的數量，即表示某個

節點可以影響到的節點，或被影響的節點數量。換言之，degree centrality 就是在計算一個人的朋友數，但 degree centrality 為 local measure，因為 degree 只看某一個節點的朋友數，表示只對外評估了第一層的節點，因此，在同一個 community 中視為重要，但在整體的網絡當中，可能會有好幾個不同的 community，此時 degree centrality 就看不出對於其他 community 的重要性。

Betweenness centrality 所表示的精神為兩個節點之間的溝通一定會透過某個節點，此時稱之此節點的 betweenness centrality 很高。其計算方式會利用到最短路徑的演算法，表示必須計算網絡中所有節點與其他點的最短路徑，因此 betweenness centrality 不同於 degree centrality，為一種 global measure。其計算方式為利用 j 到 k 節點的所有路徑中，必須經過 i 節點的數量，除以 j 到 k 節點的最短路徑數量，即可求得 betweenness centrality。下式，C_B代表 betweenness centrality，i表示為節點，$g_{jk}(i)$表示節點j與節點k之間通過節點i的最短路徑數量。公式表達如下:

$$C_B(i) = \sum_{j<k} \frac{g_{jk}(i)}{g_{jk}} \tag{1}$$

Closeness centrality 則是利用某個節點到其他點的平均最短距離，來量化一個點到其他點之間的距離。由於計算方式須求得最短距離的平均值，必須評估網絡中所有的節點距離，因此 closeness centrality 也為一種 global measure。下式，C_C代表 closeness centrality，i表示為節點，$d(i,j)$表示節點i與節點j之間平均最短距離，相加後取倒數表示量化兩個點之間的距離。公式表達如下:

$$C_C(i) = \left[\sum_{j=1}^{N} d(i,j)\right]^{-1} \tag{2}$$

本研究針對 follow 與 pull request 兩種角度所產生的網路，分析此三項指標的結果，找出最具重要性的使用者。一般來說此三種指標的結果會呈現正相關的影響，所以除了分析出重要性的使用者外，可更進一步的觀察在網路中這三種指標相互呈現負相關的情況[13]。例如當某人 degree 很高但 closeness centrality 卻很低，此特殊情況可分成六種，如表 6、表 12。本研究對於指標數值高低的定義，首先以觀察數值高的指標為基礎，將此指標從高到低排序，取出前十筆資料，再將另一指標從低到高排序後給予排名，取得排名前三筆最低數值者，如表 7 至表 11 所示，並以此方法定義為數值高低的門檻值。因此，本章節著重於在 follow 與 pull request 兩種網絡是否存在此六種情況，並觀察 GitHub 上具有此六種情況的使用者各代表的涵義。

表 6:Degree、Closeness、Betweenness on Follow Event

follow event	low degree	low closeness	low betweenness
high degree	--	(3)在自己所參與的專案(組織)內很活躍，追隨的人很多，但僅限於專案內，很難被專案外的人追隨。	(5)追隨的人很多，但跟他同類型的人很多(替代性高)，大家不會看他追隨誰而跟進，反而透過其他人(表示影響力不大)。
high closeness	(1)追隨的人不多，但跟大家的距離很近，因此很容易產生間接追隨的情況，表示很容易傳遞追隨的訊息。表示此人追隨某人後，馬上很多人就會知道。	--	(6)表示跟大家距離很近，很容易傳遞追隨的訊息，但周圍的朋友跟大家距離也都很近(很多管道)，因此大家不會看他追隨誰而跟進。
high between-ness	(2)追隨的人不多，但大家都看他追隨誰而跟進，為不同群體間的橋樑(表示影響力很大)。	(4) 此情況較少見。跟大家距離很遠，不容易傳遞追隨的訊息，但某些時候大家還是會看他追隨的人而跟進。	--

(1)觀察資料集中具有 degree 很低,但 closeness 卻很高的使用者(low degree and high closeness)。此類的使用者雖然追隨的人不多，但跟大家的距離很近，因此，很容易產生間接追隨的情況，表示很容易傳遞追隨的訊息。表示此人追隨某人後，馬上很多人就會知道。

表 7:Low Degree and High Closeness on Follow Event

使用者	degree	closeness
Hashcode	1	7.458782
Keijiro	1	6.879835
Vogella	1	6.461953

(2) 觀察資料集中具有 degree 很低，但 betweenness 卻很高的使用者(low degree and high betweenness)。此類的使用者雖然追隨的人不多，但大家都看他追隨誰而跟進，為不同群體間的橋樑，表示此類使用者影響力很大。但資料集中並未觀察出具有此特殊情況的使用者。

(3)觀察資料集中具有 closeness 很低，但 degree 卻很高的使用者(low closeness and high degree)。此類使用者所追隨的人都聚集於同一群體，在自己的群體內很活躍，但卻不易追隨其他群體，或與其他群體互動。

表 8:Low Closeness and High Degree on Follow Event

使用者	degree	closeness
Hcilab	2182	1.39169
Csjaba	1826	1.463368
Equus12	1723	1.544244

(4) 觀察資料集中具有 closeness 很低，但 betweenness 卻很高的使用者(low closeness and high betweenness)。此種情況較於罕見，此類使用者跟大家距離很遠，不容易傳遞追隨的訊息，但某些時候大家還是會看他追隨的人而跟進。

表 9:Low Closeness and High Betweenness on Follow Event

使用者	closeness	betweenness
Hcilab	1.39169	994313.9
Equus12	1.544244	822958.7
Csjaba	1.463368	687253

(5) 觀察資料集中具有 degree 很高，但 betweenness 卻很低的使用者(high degree and low betweenness)。此類使用者雖然追隨的人很多，但群體中跟他同類型的人很多，表示替代性高，大家不會看他追隨誰而跟進，反而透過其他人，表示此類使用者在群體中影響力不大。

表 10:High Degree and Low Betweenness on Follow Event

使用者	degree	betweenness
Fordream	743	0
Wycats	495	0
Jashkenas	468	0

(6) 觀察資料集中具有 closeness 很高，但 betweenness 卻很低的使用者(high closeness and low betweenness)。此類使用者雖然大家距離很近，很容易傳遞追隨的訊息，但周圍的朋友跟大家距離也都很近，可參考的管道變多，因此大家比較不會看他追隨誰而跟進。

表 11:High Closeness and Low Betweenness on Follow Event

使用者	closeness	betweenness
Hashcode	7.458782	0
Keijiro	6.879835	0
Vogella	6.461953	0

下表 12 針對 pull request 事件討論 degree、closeness、betweenness 三種指標分別呈現負相關等六種特殊情況，例如 degree 很高但 closeness centrality 卻很低，或是 betweenness centrality 很低但 closeness centrality 卻很高等，其中每個情況所代表的涵義如下表，並針對每種情況觀察出目前資料集中是否有此情況產生。

表 12:Degree、Closeness、Betweenness on Pull Request Event

pull request event	low degree	low closeness	low betweenness
high degree	--	(3)貢獻很多，但都貢獻給專案(組織)內的人。很難貢獻給專案以外的人。	(5)貢獻很多，但跟他同類型的人很多(替代性高)，大家不會透過他產生間接貢獻，而都透過別人(表示影響力不大) 。
high closeness	(1)貢獻不多，但跟大家的距離近，因此很容易產生間接貢獻。	--	(6) 與大家的距離近，但周圍的朋友跟大家距離也都很近(很多管道)，大家不會透過他產生間接貢獻。
high betweenness	(2)貢獻不多，但要貢獻時必先透過他。為不同群體間的橋樑，因此通常產生不同群體間的貢獻。	(4)此情況較少見。跟大家距離很遠，不易產生間接貢獻。但某些時候大家還是都透過他產生對另一個人的貢獻。	--

(1) 觀察資料集中具有 degree 很低，但 closeness 卻很高的使用者(low degree and high closeness)。此類的使用者雖然貢獻不多，但跟大家的距離很近，因此，很容易產生間接貢獻。

表 13:Low Degree and High Closeness on Pull Request Event

使用者	degree	closeness
Mattupstate	1	2
Ktmud	1	2
Marksteve	1	2

(2) 觀察資料集中具有 degree 很低，betweenness 卻很高的使用者(low degree and high betweenness)。此類使用者貢獻不多，但為不同群體間的橋樑，通常產生不同群體間的貢獻，此類使用者影響力很大。但資料集中並未觀察出具有此特殊情況。

(3) 觀察資料集中具有 closeness 很低，但 degree 卻很高的使用者(low closeness and high degree)。此類使用者所貢獻很多，但貢獻的人都聚集於同一群體，不易貢獻給群體以外的人。

表 14:Low Closeness and High Degree on Pull Request Event

使用者	degree	closeness
Mxcl	398	0
Antirez	39	0
Chriseppstein	37	0

(4) 觀察資料集中具有 closeness 很低，但 betweenness 卻很高的使用者(low closeness and high betweenness)。此種情況較於罕見，表示此類使用者跟大家距離很遠，不易產生間接貢獻。但某些時候大家還是都透過他產生對另一個人的貢獻。

表 15:Low Closeness and High Betweenness on Pull Request Event

使用者	closeness	betweenness
Kennethreitz	1	78
Mojombo	1	51
Mitsuhiko	1.5	26

(5) 觀察資料集中具有 degree 很高，但 betweenness 卻很低的使用者(high degree and low betweenness)。此類使用者貢獻很多，但跟他同類型的人很多，替代性高，大家不會透過他產生間接貢獻，而都透過別人，表示此人在群體中影響力不大。

表 16:High Degree and Low Betweenness on Pull Request Event

使用者	degree	betweenness
Mxcl	398	0
Antirez	39	0
Chriseppstein	37	0

(6) 觀察資料集中具有 closeness 很高，但 betweenness 卻很低的使用者(high closeness and low betweenness)。此類使用者雖然與大家的距離近，但周圍的朋友跟大家距

離也都很近，可貢獻的管道變多，因此，大家比較不會透過他產生間接貢獻。

表 17:High Closeness and Low Betweenness on Pull Request Event

使用者	closeness	betweenness
Mattupstate	1	2
Ktmud	1	2
Marksteve	1	2

3.3.2 Network Resilience 指標分析

eigenvector centrality[14]為 degree centrality 的一種變形，在 degree centrality 中我們利用某節點的所連結相鄰節點數量來評估其重要性，但這些相鄰節點都賦予相同的價值。但實際情況並非如此，在重要性的評估上，應該會根據節點本身的重要性而給予不同的價值，例如:某使用者只認識一位朋友，但朋友的身分崇高，那此使用者本身的重要性也會提高。換言之，eigenvector centrality 的精神為節點本身的重要性，取決於相鄰節點的重要性。下式，$M(v)$為一個集合，代表節點v所有相鄰的節點，λ表示為一常數，通常表示為在網絡中最大的特徵向量值，$a_{v,t}$表示節點v有連結到節點t，而在計算x_t時會再乘上$a_{v,t}$，表示節點t會受到周圍節點v的影響，亦表示節點t的重要性取決於節點v，而後在$a_{v,t}$所產生之相鄰矩陣中求得所有特徵值，並且取得最大特徵值做為λ值，使得該特徵向量為正。本研究設定 Gephi 工具 iterations 參數，此參數意義為迭代次數，表示重複執行的回合次數，以求得最後數據收斂之結果。公式表達[17]如下:

$$x_v = \sum_{t \epsilon M(v)} \frac{1}{\lambda} \quad x_t = \frac{1}{\lambda} \sum_{t \epsilon G} a_{v,t} x_t \tag{3}$$

[17] https://en.wikipedia.org/wiki/Centrality#Eigenvector_centrality

如上述所說，社群網絡分析中常利用 eigenvector centrality 分析出最具影響力的節點。因此，本小節我們透過 eigenvector centrality 指標觀察出 GitHub 上網絡結構的彈性，透過刪除這些具影響力的節點，觀察對網絡所造成的影響，稱之 network resilience[16]。一個公司組織中，若網絡結構越缺乏彈性，表示組織若遭遇組織變動或關鍵人物離職後，對組織的衝擊越大。換言之，在 GitHub 上，若網絡結構越缺乏彈性，表示 GitHub 上的合作關係越顯得脆弱，若那些具有影響力的成員不再使用 GitHub，則對 GitHub 的衝擊會相對明顯。

在 pull request 所形成的網絡中，我們試著拿掉三位關鍵人物，分別為 Mxcl、Mojombo、Kennethreitz 三位使用者，即刪除 eigenvector 指標最高的三個節點，再觀察網絡的變化。圖 16 所示，刪除關鍵人物後，網絡的平均 degree 為 0.41，中位數為 0，標準差為 0.4±0.6。而以社群網絡的角度觀察，本研究利用 Gephi 工具進行實驗的輔助，隨後設定工具中 iterations 參數值，實驗數組 iterations 參數值，調整值約為 1 到 1000，最後將 iterations 參數值設定為 320，其變化總和數值調整至小於 0.001。如下圖 17 顯示，網絡所產生的 modularity 數量變多，造成很多人被拆成單一個體，並與其他人之間都不存在連結。換言之，以 pull request 觸發事件來說，如果刪除關鍵人物，會造成 GitHub 上的貢獻減少。其主要原因為，目前 GitHub 上面使用者普遍都針對某一專案或使用者進行貢獻，很少有使用者能進行跨專案的貢獻，所以導致一旦刪除關鍵人物後，GitHub 上的協同合作關係[18]會大為遞減。換言之，目前 GitHub 上的合作關係略嫌薄弱。

反觀 follow 觸發事件所形成的網絡，圖 18 所示，刪除關鍵人物後，網絡的平均 degree 為 27.93，中位數為 13，標準差為 27.9±67.6。而以社群網絡的角度觀察，本研究利用 Gephi 工具進行實驗的輔助，隨後設定工具中 iterations 參數值，實驗數組 iterations 參數值，調整值約為 1 到 1000，最後將 iterations 參數值設定為 600，其變化總和數值調整至小於 0.01。然而研究發現，刪除關鍵人物後對於網絡結構的衝

擊相對小很多，modularity 數量沒有太大的變化。表示 follow 事件所形成網絡是一個具有彈性的網絡結構，在 GitHub 上如果關鍵人物不再使用 GitHub，其他人仍可容易的與其他人溝通互動，表示那些關鍵人物並不影響 GitHub 上的 follow 行為。

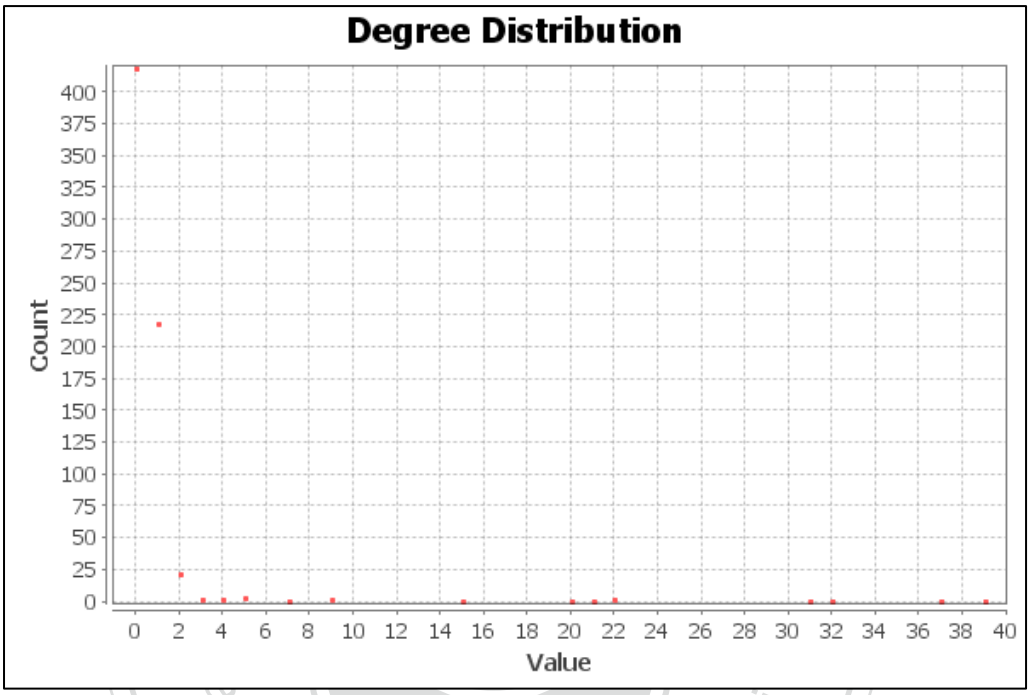

圖 16:Pull Request Degree Distribution with delete Key Point

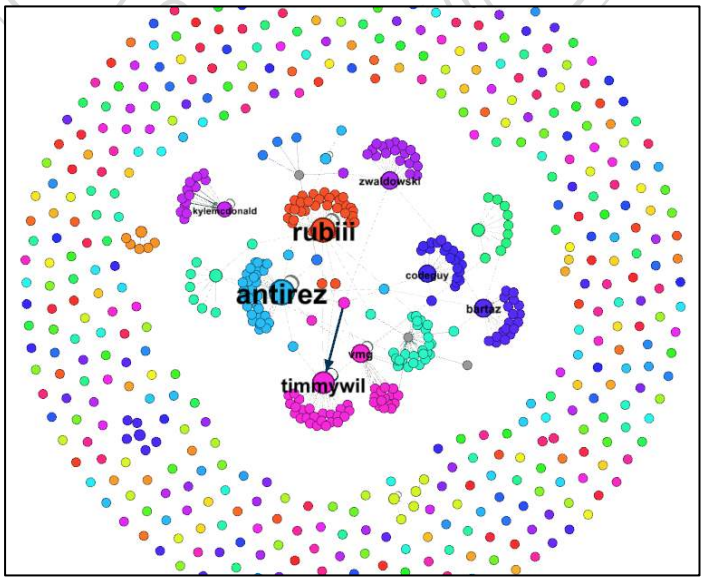

圖 17:Pull Request Modularity with delete Key Point

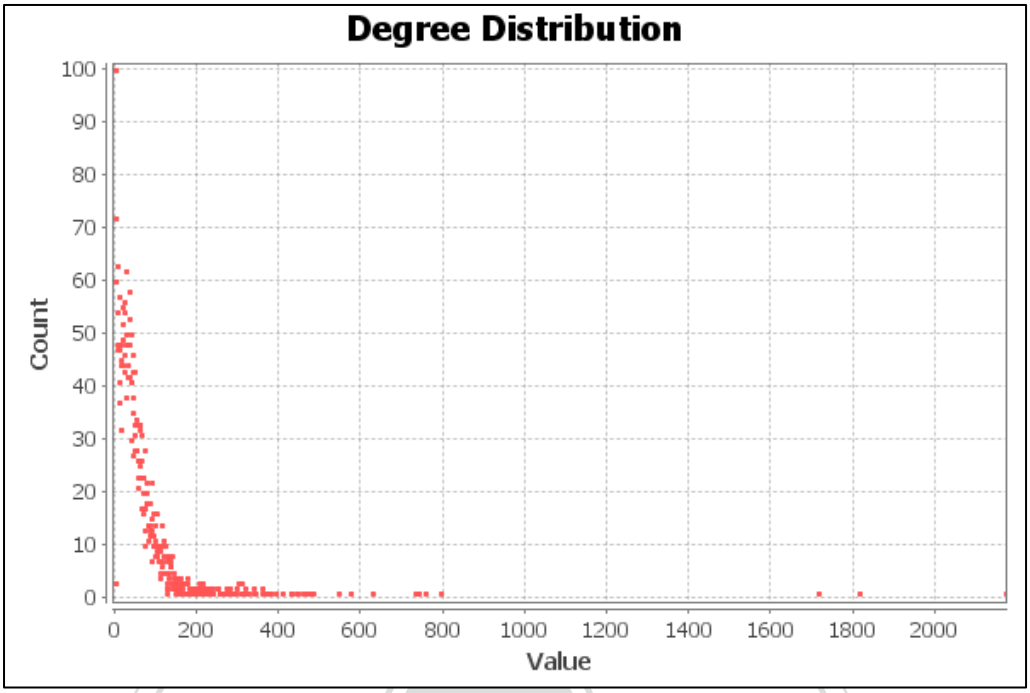

圖 18:Follower Degree Distribution with delete Key Point

下表 18 顯示 pull request 和 follow 兩個事件，經過刪除關鍵人物後，modularity 指標的變化。表中顯示 pull request 事件在經過刪除後，modularity 數量從 16 個提升至 435 個，表示有大量的使用者都變單一個體，影響了整體網絡結構。反觀 follow 事件 modularity 數量從 11 個提升至 13 個，對於整體的網絡並無太大影響。

表 18:刪除具影響力節點後 Modularity 指標的變化

	pull request		follow	
	delete before	deleted	delete before	deleted
modularity	0.568	0.797	0.351	0.360
modularity with resolution	0.473	0.700	0.297	0.305
number of communities	16	435	11	13

3.3.3 Degree Correlations 指標分析

本小節所要探討的是社群網絡分析中 degree correlations 的現象，我們透過觀察網絡中 degree 較高的那些節點，看鄰近的節點是否也為 degree 高的節點，若有此現象則

稱之具有 degree correlations。

而在 GitHub 上我們觀察那些貢獻很多的那些人，是否具有互相貢獻的情況。首先我們先取出貢獻次數最多的十位使用者，再觀察十位使用者是否有貢獻給其他九位使用者的紀錄，分析後如下圖 17 所示，結果發現此十位使用者所構成的網絡，分成七個 communities，其中五位使用者有互相貢獻情況，另外五位則為單獨個體，並未與其他九位使用者產生連結。因此，在 pull request 事件所產生的網絡中，那些貢獻度高的人中有互相貢獻的情況所占的比例不高。

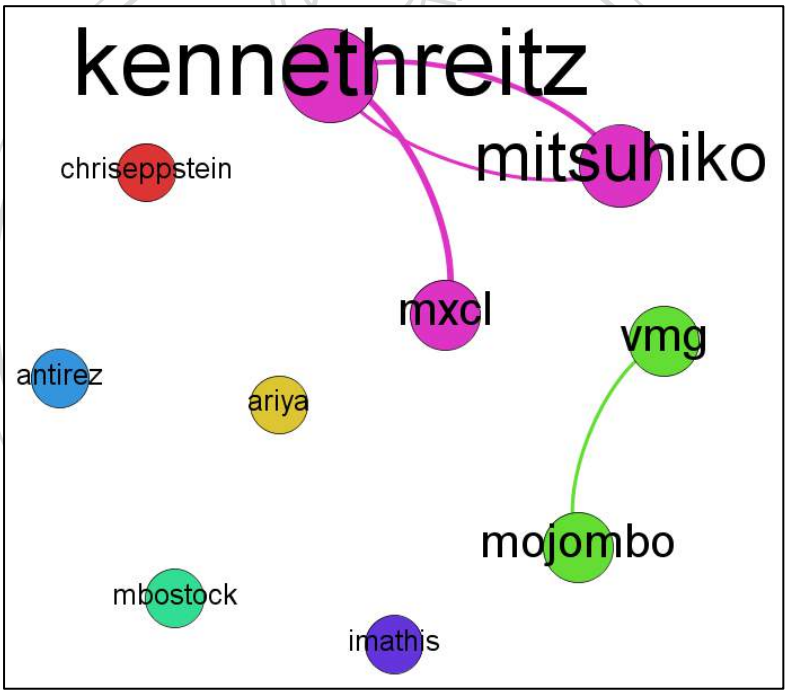

圖 19:Degree Correlations of Pull Request

第四章、資料分析

4.1 行為關聯分析

4.1.1 Correlation Coefficient 指標分析

相關係數是用來表示變數之間關係密切程度的統計指標，主要衡量兩變數間線性關聯性的高低程度，能夠明確的反應出兩個變數之間相關的程度。相關係數主要是以積差方法計算，以兩個變數與各自的平均值的差距作為基礎，透過兩變數間的差距相乘來反映兩變數間的關係程度。相關係數的值介於–1 與+1 之間，即 $-1 \le r \le +1$。當 $r > 0$ 時，表示兩變數正相關，$r < 0$ 時，兩變數為負相關。當 $|r| = 1$ 時，表示兩變數為完全線性相關，即為函數關係。當 $r = 0$ 時，表示兩變數間無線性相關關係。當 $0 < |r| < 1$ 時，表示兩變數存在一定程度的線性相關。且|r|越接近 1，兩變數間線性關係越密切；|r|越接近於 0，表示兩變數的線性相關越弱。一般可按三級劃分：$|r| < 0.4$ 為低度線性相關；$0.4 \le |r| < 0.7$ 為顯著性相關；$0.7 \le |r| < 1$ 為高度線性相關。

correlation coefficient[20]公式表達如下，其中變數 x 與變數 y 表示欲觀察其相關性的兩種變數，而變數 n 表示資料的總筆數。本研究中採用的是 pull request 事件與 follow 事件的 eigenvector 數值，並利用 Gephi 工具進行實驗的輔助，其工具設定之參數，調整參數值如 3.3.2 之說明。

$$r = \frac{n\sum xy - \sum x \sum y}{\sqrt{n\sum x^2 - (\sum x)^2}\sqrt{n\sum y^2 - (\sum y)^2}} \quad (4)$$

而本小節所研究的方向，是透過相關係數指標的分析，觀察出 pull request 事件與 follow 事件是否具有相關性。即觀察是否常常追隨別人專案的使用者是否也經常做出貢獻。透過此分析，便可觀察出目前 GitHub 上協同寫作的實際情況[19]，是否

使用者僅追隨某人而並無實質貢獻，抑或是追隨後反而會促使使用者做出貢獻。

首先，我們透過前一小節所計算的 eigenvector 指標當作基礎，即某一位使用者在 pull request 與 follow 兩種網絡中的 eigenvector 值，當成兩種欲觀察的變數，再透過相關係數指標的計算，觀察出所有使用者在 pull request 與 follow 兩個事件是否具有相關係數程度。基於前一小節 eigenvector 計算，其中我們僅取出此兩種事件 eigenvector 計算值大於零的使用者才列入分析樣本，由於在 pull request 事件中，大部分的使用者的 eigenvector 計算值皆為零，表示這些使用者在 GitHub 整體網絡中是不具影響力，因此，根據上述條件我們篩選掉不具影響力的使用者後，共取出 25 位使用者進行相關係數分析。

分析結果發現所算出來的相關係數值約為 0.13，表示 pull request 與 follow 事件之間呈現正相關，且為低度線性相關($|r| < 0.4$)。換言之，在 GitHub 上追隨者越多的使用者可能做出貢獻的機率越大。

4.1.2 PageRank 指標分析

在社群網絡分析中，最常使用 eigenvector 指標來分析網絡中節點的重要性程度。但分析的過程中必須隨著資料的特性與結構，而選用不同的分析指標。以 GitHub 資料集為例，在上一章節中我們使用 eigenvector 的數值來推算 follow 與 pull request 兩變數間的相關係數[20]，觀察是否存在正相關。而 eigenvector 指標的精神為探討在一個無方向性的相鄰矩陣中，若主要特徵向量的矩陣元素皆大於零，即可表示此節點的中心性。反觀在 GitHub 上 follow 與 pull request 兩種事件皆為有方向性的結構，例如:A 用戶追隨或貢獻給 B 用戶等方向性資料，因此，本小節使用 pagerank 指標針對有方向性的資料來進行分析。

Pagerank[21][22]指標是由 Google 的創辦人 Larry Page 所提出，主要利用網頁之間鏈結的關聯性，來評估某個網站在整個網際網路中的重要性排名，此演算法也應

用於 Google 的網頁搜尋引擎上面。其演算法主要概念為某一個網頁的重要性，會隨著所有鏈結到它的網頁之重要性而定。因此，在社群網絡分析當中也常用 pagerank 指標來觀察有方向性的資料，進而評估節點的中心性程度。延續上一章節的做法，本章節改用 pagerank 當作基礎，觀察 pull request 與 follow 兩種事件的 pagerank 值，並當作兩種觀察的變數，透過相關係數分析進行兩種事件的相關程度分析。並沿用上一章節所觀察出的 25 位使用者進行分析後，比較 pagerank 與 eigenvector 兩種指標在相同的資料下是否會呈現不同的結果。

本研究利用 Gephi 工具進行實驗的輔助，其工具設定之 pagerank 參數 probability 與 epsilon。其中 probability 參數指的就是 pagerank 演算法中的 damping factor，稱之延滯係數或阻尼常數，其涵義為使用者到達某頁面後，繼續往後瀏覽的機率，也表示是使用者停止瀏覽，隨機轉到新的頁面(URL)的機率。本研究設定 probability 為 0.85，此數值也為目前 pagerank 演算法所建議使用之參數預設值，而本研究實驗調整參數值從 0.1、0.5、0.85 至 1.0，來觀察 pagerank 數值的變化，發現數值的大小會隨著參數值越大而變大。

另外，epsilon 參數表示為 pagerank 充分收斂時所有節點的 pagerank 數值的最小累積變化，其涵義則為收斂的停止條件。本研究設定 epsilon 參數值為 0.001，當作收斂的停止條件，而本研究實驗調整參數值從 1、0.1、0.01、0.001 等，以此類推，研究發現設定至 0.001 之後 pagerank 數值將為穩定不變，表示此數值已收斂。而觀察 pagerank 數值的變化，發現數值是隨著參數值越小，而 pagerank 數值會越大，直到收斂後數值不再變化為止。

計算出收斂過後的 pagerank 值後，本研究後續觀察 pull request 與 follow 兩種事件的 pagerank 值，分析結果發現，所計算出來的相關係數為-0.06，表示 pull request 與 follow 事件之間呈現微弱相關，且絕對值介於$|r| < 0.4$ 為低度線性相關。也可發現 pagerank 與 eigenvector 兩種指標呈現了完全不同的結果，在 eigenvector 指標中，兩

種事件呈現正相關關係，而改用 pagerank 後卻呈現的微弱相關。因此，在社群網絡分析中，必須因應不同的資料結構來選取適當的指標來進行觀察，因為不同的指標可能會造成資料分析結果的差異。以本研究為例，在 4.1.1 小節中使用 eigenvector 指標所觀察出 pull request 與 follow 事件之間呈現正相關，其相關係數值約為 0.13；而在 4.1.2 小節中使用 pagerank 指標，pull request 與 follow 事件之間卻呈現微弱相關，其相關係數為-0.06，使用兩種不同指標觀察，卻獲得完全相反的結論。

以上結論與參考論文中 Coding Together at Scale: GitHub as a Collaborative Social Network[18]所提到的結論相反，文中提到 follow 將有助於 collaborative，此論文中使用斯皮爾曼相關係數[23]觀察 follow 與 collaborative 之間的相關性，而結論顯示此兩者之間為正相關，相關係數約為 0.257。由於先前使用的是皮爾森相關係數[20]，而參考論文所使用者是斯皮爾曼相關係數，因此，本研究再針對 pagerank 指標做斯皮爾曼相關係數的計算，結論得出 follow 與 collaborative 之間為正相關，相關係數約為 0.645，結論亦與參考論文相同。

4.2 專案貢獻度

本小節所要討論的是 GitHub 上每位使用者的貢獻程度[6]，首先計算出每個帳戶的總貢獻次數後，除以帳戶的總數量，取得每個帳戶的平均貢獻次數。由於帳戶又區分為兩種不同型態，一種為一般使用者，另一種為組織，資料集中以 type 欄位區分為 user 與 organization 兩種，因此，分開計算此兩種型態的平均貢獻程度。

首先，計算所有 user 貢獻總次數，共有 74,521 筆， 除以有貢獻的 user 總人數 19,517 人，得出每人平均貢獻次數約為 3.82 次。另外，帳戶型態為 organization 的總貢獻次數為 4,324 筆，除以有貢獻的 organization 總數，共有 713 個，得出每個組織平均貢獻次數約為 6.06 次。

計算平均貢獻度後，接著計算沉睡用戶指標，其指標定義為使用者只貢獻過一

次，或時間間隔很久才貢獻一次的使用者。首先我們先定義怎樣的情況算是隔了很久才貢獻，因此，我們算出每位使用者每一次到下一次貢獻的時間差距，算出每位使用者平均多久會貢獻一次，其中時間區間以分鐘數作為計算，若超出平均貢獻時間者，則定義為沉睡用戶。計算結果顯示一般使用者平均 27 天貢獻一次，組織則平均 12 天貢獻一次，組織的貢獻頻率較一般使用者來的頻繁，之所以有如此差異是由於 GitHub 上型態為 organization 的帳戶可由多位組織成員進行使用，即表示有多位成員同時進行貢獻，因此，組織帳戶的平均貢獻天數通常會小於一般使用者帳戶。user 與 organization 的平均貢獻時間如下表:

表 19:User 與 Organization 的平均貢獻

型態\時間差	分鐘	小時	天數
user	39,150.79	652.51	27.19
organization	17,692.26	294.87	12.29

其中有 5,728 位 user 超過平均貢獻時間，即有 5,728 個 user 位沉睡用戶，將沉睡用戶除以 user 總人數，得出沉睡用戶約佔總數的 29% (5728/19517=0.29)。而 organization 方面，有 197 個組織超過平均貢獻時間，即有 197 個組織為沉睡用戶，除以組織總數，得出沉睡用戶約佔總數的 28% (197/713=0.28)。

4.3 專案吸引力與黏著度

本小節以專案角度去觀察各個專案質量的評估，分成專案的吸引力與黏著度兩個層面做研究與討論。透過專案的吸引力的觀察，便可得知目前 GitHub 上專案發展的趨勢，看使用者是否漸漸的開始注意到此專案，而是否有可能會成為下一個熱門的技術。反之，透過專案黏著度的觀察，便可得知一個興起的專案是否具有永續發展性，看使用者是否對於此專案具有忠誠度[8]，持續的進行程式碼貢獻，讓此專案得以更

加活躍與完善。

首先,我們先針對專案的吸引力與黏著度做出定義,我們定義在資料集中的 fork 事件作為吸引力評估的基礎,透過計算每個專案每年 fork 的成長率來算出各專案的吸引力數值。即今年某專案被 fork 的數量(等於 fork 此專案的使用者數量),比較去年此專案被 fork 的數量,進行成長率的計算。最後即可得知每個專案是否具有吸引力特質,並且算出吸引力比例。計算公式如下:

$$\frac{(\text{今年 fork 數量} - \text{去年 fork 數量})}{\text{去年 fork 數量}} \tag{5}$$

而專案黏著度方面,我們以資料集中的 pull request 事件作為黏著度評估的基礎,定義各專案中每個年度做出貢獻的開發者,在下個年度是否仍然做出貢獻,並以此進行各年度間比例的計算。計算公式如下:

$$\frac{\text{專案今年度貢獻者總數}}{\text{專案下一年度持續貢獻者總數}} \tag{6}$$

計算方式為專案 A 在 2011 年有 3 位開發者,2012 年有 2 位持續貢獻程式碼,所計算的比例則為 2/3。專案 B 在 2012 年有 1 位開發者,2013 年有 1 位持續貢獻程式碼,所計算的比例為 1/1。以下說明研究實際做法,吸引力方面先將資料集中各專案在各年被 fork 的數量做加總,其中資料區間為 2008 到 2013 年間,且過濾首年的資料,原因為首年資料因為缺乏去年資料,而無法進行成長率的計算。在過濾首年資料後,再針對各年度的成長率進行平均,計算出各專案的平均成長率,如下圖 20 所示。

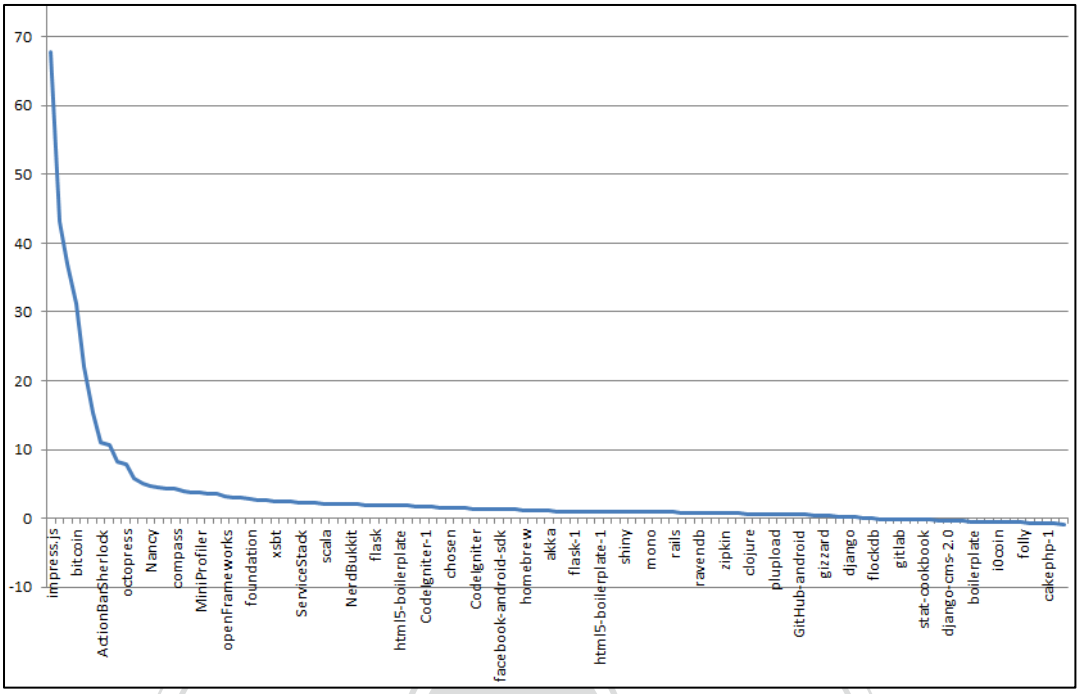

圖 20:專案吸引力平均成長率

透過以上分析即可顯示出目前 GitHub 上目前哪幾個專案近幾年呈現逐漸成長的趨勢，平均成長率呈現正數成長，也可觀察出是否有哪幾個專案目前也呈現衰退的情形，數值呈現負成長，利用此數據便可讓使用者進行決策是否專注於某個專案的開發。而專案黏著度方面，由於資料集中 pull request 事件的資料區間集中於 2010 年到 2013 年，所以我們先分別計算 2010 至 2011、2011 至 2012、2012 至 2013 年三個區段的黏著度比例。即計算 2010 年度各專案的貢獻者，到 2011 年度是否有繼續進行貢獻。最後再算出 2010 年到 2013 年間各專案黏著度的變化，是趨於成長或緩降，如下圖 21 所示。

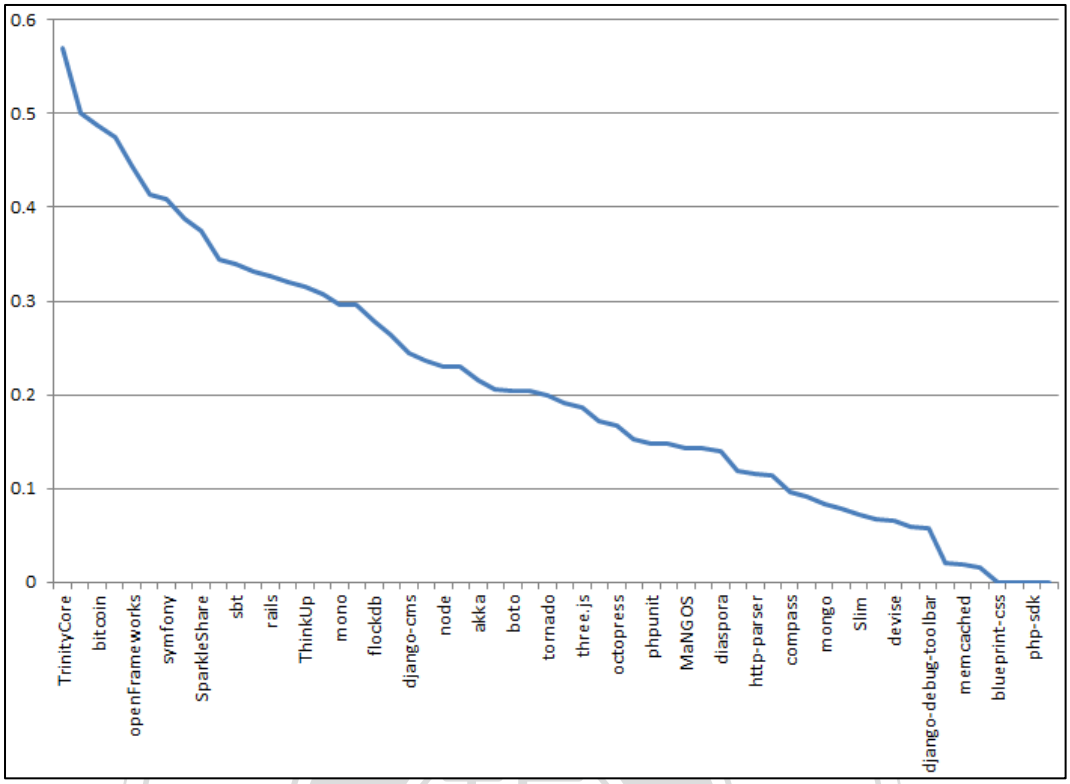

圖 21:專案黏性平均成長率

最後，我們將吸引力與黏著度兩種指標做資料正規化，將資料重新分布至同一水平下做比較。談到資料正規化，大致上會用以下三種方法，(一)極值正規化 Min-max normalization，(二)Z-分數正規化 Z-score normalization，(三)十進位正規化 Normalization by decimal scaling。第一種方法適用於將資料規範在某一個指定範圍內的情況，第二種則適用於利用平均值作為基準，觀察高於或低於平均值的情況，第三種則適用於將數字壓縮到區間 0 到 1 的情況。然而以本研究的資料集而言，適合將資料壓縮到 0 到 1 之間去進行吸引力與黏著度的比較，並利於在圖表呈現吸引力與黏著度的高低，因此，本研究採用了 S.Gopal Krishna Patro 正規化[15] (normalization by decimal scaling)來進行資料正規化，十進位正規化表達式如下，其中 v 值為欲正規化的數值，new_v 為正規化後的數值，c 值為使得max(|new_v|) < 1的最小整數。公式表達如下:

$$\text{new\_v} = \frac{v}{10^c} \tag{7}$$

Where c is the smallest integer such that max(|new_v|) < 1

隨後並轉換成二維圖形，X 軸放置黏著度指標，Y 軸放置吸引力指標。透過專案吸引力我們可以知道哪個專案目前持續有人在進行貢獻，不至於有衰退的情形，而專案黏著度的成長或下降則是能夠表示出專案的永續發展程度，可讓使用者去評估是否需要再持續針對某個專案進行貢獻。最後，我們透過每個專案的吸引力與黏著度數值，分析出如下圖 22 所示，並利用此圖觀察出目前 GitHub 上各個專案是屬於哪種性質(吸引力或黏著度)居多。

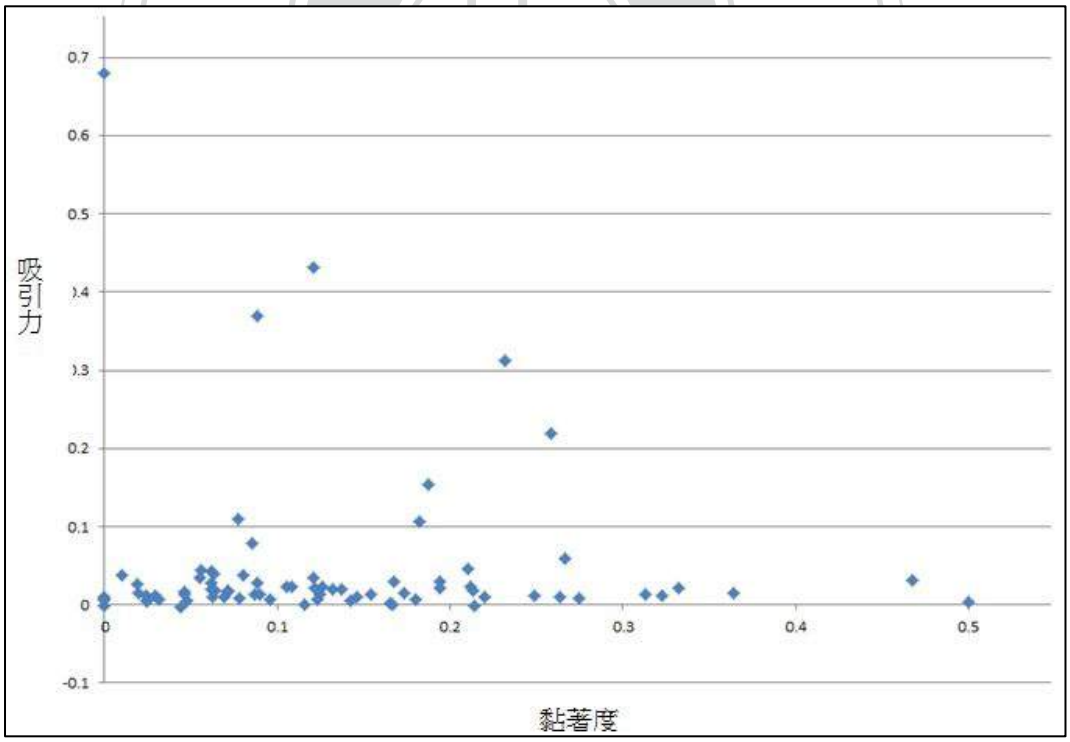

圖 22:專案吸引力與黏著度分佈

4.4 專案演進

本章節所要探討的主題為專案的演進[4]，分析專案的活動狀態是否會隨著時間而有所改變，藉此便能夠進一步了解此專案的未來走向。首先我們參考 Kazuhiro Yamashita、Shane McIntosh、Yasutaka Kamei 和 Naoyasu Ubayashi 等四位作者所共同撰寫的論文[11]，將專案活動的狀態分成四種型態，分別為活躍期、流動期、穩定期與衰退期。本研究利用上一章節專案吸引力與黏著度的分析資料，並以平均值做為門檻值，計算各年度的平均成長率後，將高於平均值以上定義為高度吸引力或高度黏著度，低於平均值以上則定義為不具吸引力或低黏著度。以此類推，便可進一步定義以下四種專案型態，定義如下表 20。

表 20:專案型態定義表

活躍期	具有高度吸引力，且高度黏著度(持續貢獻)的專案。 吸引多數人下載專案，且現有的開發人員仍持續貢獻
流動期	具有高度吸引力，但低黏著度的專案。 吸引多數人下載專案，但沒成功留住現有的開發人員
穩定期	已不具吸引力，但卻具高度黏著度的專案。 留住現有的開發人員，但卻很難吸引到新成員
衰退期	已不具吸引力，也不具黏著度的專案。 很難留住現有的開發人員，也沒有吸引到新成員。

定義專案活動的四種型態後，再將專案的吸引力與黏著度資料依照年度區分，以本研究為例，採取資料區間為 2011 到 2013 年度資料。並分析出各年度的各個專案所屬型態，資料庫計算中以 fstat='high' and pstat='high'表示為高吸引力、高黏著度專案，並標註為 A(活躍期)。以 fstat='high' and pstat='low' 表示為高吸引力，低黏著度專案，並標註為 B (流動期)。以 fstat='low' and pstat='high' 表示為低吸引力，高黏著度專案，並標註為 C (穩定期)。以 fstat='low' and pstat='low' 表示為低吸引力，低黏著度專案，並標註為 D(衰退期) 。並利用以下語法計算出各專案所處的型態。

```
SELECT * from (
   SELECT repo,
   case fstat when 'low' then case when pstat ='low' then 'D' else 'C' end
   when 'high' then case when pstat ='high' then 'A' else 'B' end end 2011s
   FROM `20150522_fork_pull_2011`
) as a
inner join
(
   SELECT repo,
   case fstat when 'low' then case when pstat ='low' then 'D' else 'C' end
   when 'high' then case when pstat ='high' then 'A' else 'B' end end 2012s
   FROM `20150522_fork_pull_2012`
) as b on a.repo = b.repo
inner join
(
   SELECT repo,
   case fstat when 'low' then case when pstat ='low' then 'D' else 'C' end
   when 'high' then case when pstat ='high' then 'A' else 'B' end end 2013s
   FROM `20150522_fork_pull_2013`
) as c on a.repo = c.repo
```

表 21:專案演進型態對照表

專案名稱	2011	2012	2013
clojure	衰退	流動	衰退
compass	流動	流動	衰退
django-cms	活躍	衰退	活躍
django-debug-toolbar	衰退	衰退	衰退
jquery	流動	流動	流動
memcached	衰退	流動	流動
paperclip	衰退	衰退	衰退
rails	活躍	活躍	活躍

承接上述資料，分析出 GitHub 上各專案在 2011 到 2013 年度的演進狀態，如上表 21 所示，後續本研究更進一步的將此資料計算出各個狀態之間轉換的機率，進而能夠評估出目前 GitHub 上的專案狀態轉換的一個過程，資料庫計算語法如下所示:

```
SELECT
    (select count(*) as co1 FROM `20150531_fork_pull_projecttime`
    where 2011s=' A ' and 2012s='A') as a,
    (select count(*) as co2 FROM `20150531_fork_pull_projecttime`
    where 2012s=' A ' and 2013s='A') as b,
    (select count(*) as co1 FROM `20150531_fork_pull_projecttime`
    where 2011s=' A ' and 2012s='B') as c,
    (select count(*) as co2 FROM `20150531_fork_pull_projecttime`
    where 2012s=' A ' and 2013s='B') as d,
    (select count(*) as co1 FROM `20150531_fork_pull_projecttime`
    where 2011s=' A ' and 2012s='C') as e,
    (select count(*) as co2 FROM `20150531_fork_pull_projecttime`
    where 2012s=' A ' and 2013s='C') as f,
    (select count(*) as co1 FROM `20150531_fork_pull_projecttime`
    where 2011s=' A ' and 2012s='D') as g,
    (select count(*) as co2 FROM `20150531_fork_pull_projecttime`
    where 2012s=' A ' and 2013s='D') as h
FROM `20150531_fork_pull_projecttime`
```

分母為各型態轉換的總次數，專案型態 A 轉換到專案型態 A 為 15 次，型態 A 轉換到型態 B 為 5 次，型態 A 轉換到型態 C 為 7 次，依序加總後分母為 110。接下來計算各階段型態轉換之機率，型態 A 轉換到型態 A，機率為 15/110 = 0.136(13.6%)。型態 A 轉換到型態 B，機率為 5/110 = 0.045(4.5%)。型態 A 轉換到型態 C，機率為

7/110 = 0.064(6.4%)。型態 A 轉換到型態 D，機率為 3/110 = 0.027(2.7%)。型態 B 轉換到型態 A，機率為 4/110 = 0.036(3.6%)，以此類推求得各階段轉換之機率。

以上述方式將 2011 到 2013 年所有專案狀態的變化進行統計，首先四種狀態共有十六種排列組合，再分別計算這十六種組合在各年間出現的次數，最後再除以十六種組合的總次數，以求得各狀態轉換的機率值。我們參考 Kazuhiro Yamashita、Shane McIntosh、Yasutaka Kamei 和 Naoyasu Ubayashi 等四位作者所共同撰寫的論文[11]，將各狀態轉換的機率值繪製成圖 21。如圖 21 所示，目前專案中呈現流動的狀態為最高，佔 23.6%；而呈現活躍的狀態佔次高，為 13.6%；值得注意的是目前專案流動轉衰退狀態佔了較高的機率，為 9%，且衰退狀態若要回到活躍的狀態，所佔的機率為最低，為 1.8%。因此，表示如果專案一旦處於流動狀態，便有較高的機率形成衰退；且一旦專案衰退之後，便較無可能再回到活躍的狀態。

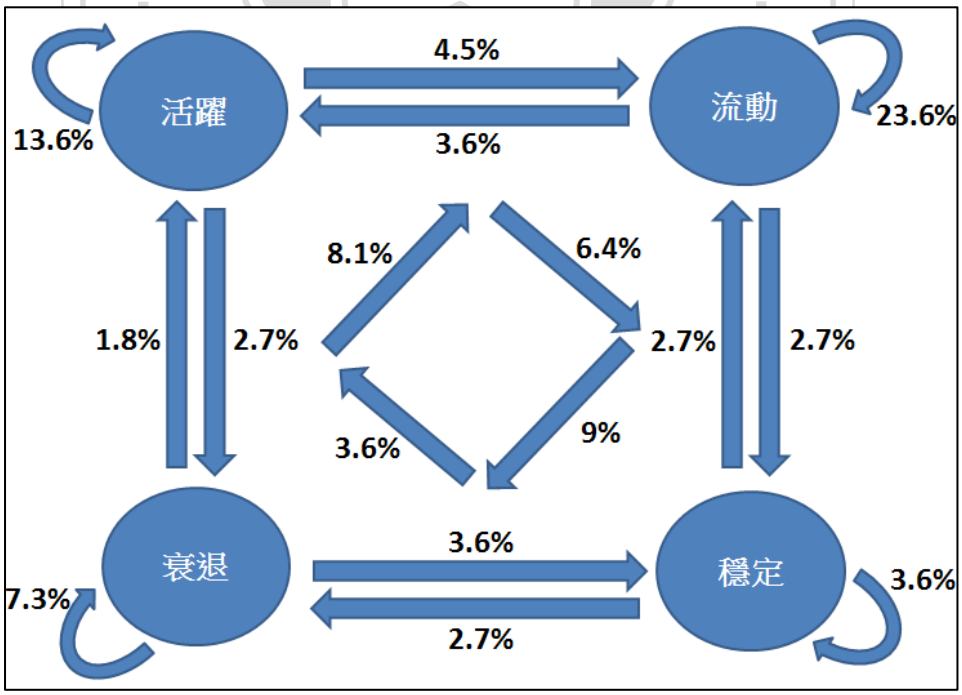

圖 23:專案演進機率[11]

第五章、研究發現與討論

本研究藉由社群網絡相關指標的觀察與自行定義的分析方法，提出以下七點發現，以下七點發現僅適用於 GitHub 相關資料結構之平台，或有相同操作模式與事件的協同寫作平台。

(一) 針對資料集的可信度進行分析與驗證，推論資料集的可信程度。本研究藉由 MSR 資料集中所觀察出專案最常使用的程式語言，與最常被關注的專案所使用的程式語言排名，並利用觀察出的結果與 GitHub 官方所釋出的排名進行比較，發現差異度不大。表示本研究所使用的資料集，能夠反映目前 GitHub 上的真實情況，以此推論本研究所採用的資料集具有一定的可信度與有效性。但與 TIOBE 程式語言排行榜的比較中發現差異較大，表示目前 GitHub 平台上所使用的程式語言，未能反映出目前多數程式開發人員所使用的主流技術。

(二) 目前 GitHub 平台上使用者僅會針對單一使用者進行追隨，造成某一使用者擁有大量追隨者，不同於一般社群網站呈現相互追隨的情形，表示各使用者間的連結性是不足的，屬於低互動的社群網絡，並且從事件總數來看，使用者進行貢獻的次數為所有事件中最少，也表示目前 GitHub 上呈現低貢獻的情況。研究一開始針對資料的結構分析，統計平台上各個事件的數量，發現使用者仍然以觀看專案當作主要用途，而少數人會進行專案的參與。之後進行社群網絡分析，研究將有共同提交不同專案的使用者，並且針對相關指標分析，例如平均 degree 與 degree 分佈、network resilience 與 degree correlations，發現目前 GitHub 平台上鮮少有同時對多個專案進行提交的貢獻者，且 pull request 事件的 degree 分佈遠低於 follow 事件的 degree 分佈，表示每個人平均貢獻度其實不高，而後針對網絡彈性方面的分析，也發現 follow 事件所形成網絡是一個具有彈性的網絡結構，在 GitHub 上如果關鍵人物不再使用 GitHub，其他人將不受影響;反觀以 pull request 所形成的網絡則呈現一個缺乏彈性的網絡結構，表示若刪除關鍵人物，將會直接造成 GitHub 上的貢獻減少。研究最後也

針對資料集中那些關鍵人物，看是否會互相進行貢獻，結果發現那些貢獻度高的人中有互相貢獻的情況所占的比例不高。本研究也針對貢獻度進行觀察，結果發現目前 GitHub 平台上沉睡用戶約佔總數的三成，表示約有三成的人低於平均貢獻時間。

(三) GitHub 平台上呈現長尾效應現象，且最常使用的專案與使用者感興趣的專案並不一定成正相關。研究發現在 MSR 的資料集中有不少使用 Java 程式語言的專案，專案數量約為八千至九千，但針對使用者關注的專案中，卻顯示目前較少數人關注以 Java 語言為主的專案。由此可推論目前 Java 語言非目前大眾所感興趣的主流程式語言，反觀 JavaScript 與 Ruby 兩種語言不僅是專案中最常使用之語言，同時也為最多人關注的語言，兩圖表(圖四、表四)交叉比對下，更能反映目前這兩種程式為目前主流學習語言，也與大眾理解中的相符合。以整體專案來觀察，前百分之二十的專案，僅獲得 48%關注人數，反之剩餘百分之八十的專案，反而獲得 52%的關注人數，呈現一種長尾效應現象。表示 GitHub 上的使用者較不容易產生跟隨熱門專案之情形，即使較為冷門的專案也會有不少使用者進行關注。

(四) 具有高度關聯性的專案通常是相同類型的程式語言。本研究透過社群網絡分析，利用是否有使用者同時關注相同專案當作關聯，分析出專案之間的關聯性，並觀察具有高度關聯性的專案。研究觀察出專案 doom3.gpl、folly、MaNGOS 三者之間具有關聯性，且都使用 C++語言，而 stat-cookbook、ProjectTemplate、devtools、knitr 具有關聯性，且都使用 R 語言等。結果顯示具有高度關聯性的專案通常是相同類型的程式語言，表示在 GitHub 上使用者的操作行為模式，通常考慮自身領域相關的程式語言技術，在關注專案時，鮮少會關注不同領域的技術。

(五) 推論程式語言間的相關性與公司組織之間的關聯。透過社群網絡分析，可以觀察出語言之間的相關性。Scala、JavaScript、R 具有相關性，表示使用者同時會這些程式語言的機率較高。Java、Python、PHP 具有相關性，而 C、C#、C++、Ruby，也具有相關性。透過公司組織之間的關聯性觀察出兩個組織之間的合作與投資關係。

研究指出組織之間的關聯，可透過新聞事件來觀察其真實性，例如新聞指出 Facebook 與 Joyent 兩家公司具有互相合作的關係，而 Twitter 與 Netty 也同樣具有合作關係，或是 Scala 與 Akka 具有相同技術開發等事實。

(六) 追隨事件與最後是否進行貢獻之間並無顯著的關係。在本研究 4.1.2 小節中使用 pagerank 指標觀察 pull request 與 follow 事件之間的相關係數程度，推論是否具有相關性，而結果顯示此兩種事件呈現微弱相關，其相關係數為-0.06。表示 GitHub 平台上使用者追隨某人時並不會影響最後是否貢獻給此人。以上結論與參考論文中 Coding Together at Scale: GitHub as a Collaborative Social Network[18]提到 follow 將有助於 collaborative 的結論相反，此論文中使用斯皮爾曼相關係數觀察 follow 與 collaborative 之間的相關性，而結論顯示此兩者之間為正相關，相關係數約為 0.257。而後本研究再針對 pagerank 指標做斯皮爾曼相關係數的計算，結論得出 follow 與 collaborative 之間為正相關，相關係數約為 0.645，結論亦與參考論文相同。

(七) 推論 GitHub 專案的演進階段與演進機率。本研究利用 GitHub 平台上的 fork 事件計算出專案吸引力，並比較各年度吸引力的成長率；之後利用 pull request 事件作為專案黏著度評估的基礎，計算專案中各年度持續貢獻的開發者，並進行各年度間持續貢獻的比例計算。計算出專案的吸引力與黏著度數值後，將其兩種變數視為兩種維度，將其轉換成四個象限之圖形，即表示專案的四種演進的過程，分別為活躍期、流動期、穩定期與衰退期。之後透過現有 MSR 資料集推論目前 GitHub 平台上專案所呈現的實際情況，將其專案區分出專案的四種演進階段，而推論出 2011 年到 2013 年所有專案的演進趨勢。後續本研究更進一步的將此資料計算出各個狀態轉換的機率，進而能夠評估出目前 GitHub 上的專案狀態轉換的一個過程，以此分析出專案未來可能的趨勢與走向。

第六章、未來展望

本研究針對 GitHub 平台上的社群現象進行各項指標的分析，爾後延續本研究的理念，可延伸出以下四點未來研究的走向：第一、藉由程式語言或公司組織之間關聯，找出最適合的開發人選加入協同成員。本研究結論指出 GitHub 平台上使用者僅會針對單一使用者進行追隨，造成某一使用者擁有大量追隨者，不同於一般社群網站呈現相互追隨的情形，也發現 GitHub 上呈現低互動與低貢獻的情況，因此，當專案缺乏開發人員，或是想增加開發人員時，便會遇到尋找開發人員的困難，而該如何在眾多成員的 GitHub 平台上挑選適合的開發人員，讓此成員能最快的時間融入團隊開發模式與規範，這將會是相當重要的一個議題。然而透過本研究所觀察程式語言的關聯性，未來可研究某種程式語言的專案在執行時，若有開發人員不足的現象時，找出另一種關聯性高的程式語言，並找出會這種語言的開發人員，並推薦他加入此專案成員，因為這兩種語言中間有一定程度的共通性，表示會此種語言，學習另外一種語言的門檻不高;或是此兩種語言具有共存關係，會此種語言的開發人員，通常也會使用另一種語言，例如:網頁前端技術中，會 javascript 的開發人員，通常也會使用 jquery 語言等。

另外，透過觀察組織之間的關聯性，表示這兩個組織之間可能是互有投資，或是技術相似等原因，而本研究所觀察兩個組織的關聯性是以兩個組織之間具有相同的開發人員為基礎，因此，此類型的開發人員熟悉這兩種組織所屬專案的開發模式，未來可利用於尋找組織成員，推薦此人加入另一組織之專案，更進一步可利用於公司組織的招募之用途。

第二、延伸本研究所區分的四種專案演進階段，可更進一步細分為多個演進階段。在數據分析領域上，"潛在使用者"是常常會被計算的指標，透過潛在使用者分析能夠了解到市場需求的趨勢與走向，並得知目前產品的使用情況，進而調整公司產品的發展方向。然而在 GitHub 上能否分析出潛在使用者這也是另一個可研究的議

題，透過分析出潛在使用者，可了解到目前專案的發展情況，例如:本研究 4.4 小節專案演進階段，再細分出偶爾使用的使用者，或是隔一段時間突然使用的使用者，列為潛在客戶變數，分析當吸引力高、潛在客戶少，或是黏著度低、潛在客戶多等情況。本研究使用專案的吸引力與黏著度區分出專案階段的四個象限(活躍期、流動期、穩定期、衰退期)，未來可加入潛在客戶變數，列入第三種維度，便可重新定義專案階段，利用不同的專案演進階段進行更精細之分析。

第三、建立 GitHub 平台上的推薦技術，根據適合使用者的專長技術，推薦適合的開發專案。近幾年來推薦系統的發展多元，涵蓋多種領域，各式各樣的推薦主題孕育而生。所謂的推薦系統是依據使用者的喜好、興趣，或是使用行為，來推薦使用者可能會感興趣的商品或資訊。未來可利用此概念，利用在 GitHub 平台上，以本研究為例，在第四章節中，藉由分析專案演進可得知專案的興盛與衰退的過程，便可利用此資訊進行專案的推薦，同時過濾與分類使用者的專長技術，甚至是所屬組織，將目前處於活躍期的專案推薦給合適的使用者，甚至利用專案的演進推估專案未來趨勢，便可提高推薦精準度，讓使用者在 GitHub 上更容易的找到所需的資訊。

第四、GitHub 資料集中與 TIOBE 程式語言排名的差異分析。隨著近幾年數據分析的興起，在 TIOBE 中已可看出 MATLAB、R 等利於統計分析的程式語言排名漸漸上升，然而本研究在 2.5.2 小節中比較了 GitHub 資料集中與 TIOBE 的排名差異，在比較中發現 GitHub 資料集中未發現此類統計相關程式語言有成長的現象，資料集中目前仍以主流程式語言為主，如 java、ruby 等，未來研究可針對此議題進行更進一步的分析與研究，並且強調針對 GitHub 資料集上的分析，更能反應出主流的真實情況，以增進研究的可信度與研究價值。

參考文獻

[1] Fitzgerald Brian, "Software Crisis 2.0," IEEE Computer Society, pp. 91-93, 2012.

[2] Huzefa Kagdi; Michael L. Collard; Jonathan I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," Journal of Software Maintenance and Evolution: Research and Practice, pp. 77–131, 2007.

[3] João Brunet; Gail C. Murphy; Ricardo Terra; Jorge Figueiredo; Dalton Serey, "Do Developers Discuss Design?," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp.340–343, 2014.

[4] Karan Aggarwal; Abram Hindle; Eleni Stroulia, "Co-evolution of Project Documentation and Popularity within Github," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 1-4, 2014.

[5] Daniel Pletea; Bogdan Vasilescu; Alexander Serebrenik, "Security and Emotion: Sentiment Analysis of Security Discussions on GitHub," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 1-4, 2014.

[6] Rohan Padhye; Senthil Mani; Vibha Singhal Sinha, "A Study of External Community Contribution to Open-Source Projects on GitHub," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 332-335, 2014.

[7] Jyoti Sheoran; Kelly Blincoe; Eirini Kalliamvakou; Daniela Damian; Jordan Ell, "Understanding “Watchers” on GitHub," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 336-339, 2014.

[8] Emitza Guzman; David Azócar; Yang Li, "Sentiment Analysis of Commit Comments in GitHub: An Empirical Study," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 352-355, 2014.

[9] Nicholas Matragkas; James R. Williams; Dimitris S. Kolovos; Richard F. Paige, "Analysing the Biodiversity of Open Source Ecosystems: The GitHub Case," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 356-359, 2014.

[10] Mohammad Masudur Rahman; Chanchal K. Roy, "An Insight into the Pull Requests of GitHub," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 364-367, 2014.

[11] Kazuhiro Yamashita; Shane McIntosh; Yasutaka Kamei; Naoyasu Ubayashi, "Magnet or Sticky? An OSS Project-by-Project Typology," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 1-4, 2014.

[12] Linton C. Freeman, "Centrality in Social Networks Conceptual Clarification," Social Networks, pp. 215-239, 1978.

[13] Chaoqun Ni; Cassidy R. Sugimoto; Jiepu Jiang, "Degree, Closeness, and Betweenness: Application of group centrality measurements to explore macro-disciplinary evolution diachronically," Social Networks, pp. 1-10. 2011.

[14] Stephen P. Borgatti, "Centrality and network flow," Social Networks, pp. 55–71. 2005.

[15] S.Gopal Krishna Patro; Kishore Kumar sahu, "Normalization: A Preprocessing Stage," Computer Science, pp. 1-4, 2015.

[16] Mark. E. J. Newman, "The structure and function of complex networks," SIAM review, pp.1-58, 2003.

[17] Laura Dabbish; Colleen Stuart; Jason Tsay; Jim Herbsleb, "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository," In Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work, pp. 1-10,

2012.

[18] Antonio Lima; Luca Rossi; Mirco Musolesi, "Coding Together at Scale: GitHub as a Collaborative Social Network," In Proceedings of 8th AAAI International Conference on Weblogs and Social Media, pp. 1-10, 2014.

[19] Yuriy Tymchuk; Andrea Mocci; Michele Lanza, "Collaboration in Open-Source Projects: Myth or Reality?," In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 304-307, 2014.

[20] Joseph Lee Rodgers; W. Alan Nicewander, "Thirteen Ways to Look at the Correlation Coefficient," The American Statistician, pp. 59-66, 1988.

[21] Sergey Brin; Lawrence Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," In Proceedings of the seventh International Conference on the World Wide Web, pp107-117, 1998.

[22] Lawrence Page; Sergey Brin; Rajeev Motwani; Terry Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Technical report, Stanford Digital Library Technologies Project, pp.1-17, 1999.

[23] Jerrold H Zar, "Spearman Rank Correlation," Encyclopedia of Biostatistics, pp.1-6, 1988.

[24] Vincent D Blondel; Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre, "Fast unfolding of communities in large networks," in Journal of Statistical Mechanics: Theory and Experiment , pp1-11, 2008.

[25] Renaud Lambiotte; Jean-Charles Delvenne; Mauricio Barahona, "Laplacian Dynamics and Multiscale Modular Structure in Networks," IEEE Transactions on Network Science and Engineering, pp1-29, 2009.