千葉工業大学　社会システム科学部

プロジェクトマネジメント学科

平成26年度　卒業論文

テストを基準にしたOSS開発プロジェクトにおける

タスク処理過程の定量分析

Quantitative analysis of the task processing process in the OSS development project on the basis of a test

|  |  |
| --- | --- |
| 指導教員印 | 学科受付印 |
|  |  |

ソフトウェア開発管理コース

矢吹研究室

1142009　安藤 勇樹／Yuki Ando

**目次**

[**第1章** **序論**](#_Toc410263935)

[**1.1.** **本章の構成** - 1 -](#_Toc410263936)

**[1.2.](#_Toc410263937)****[研究背景](#_Toc410263937)** [- 1 -](#_Toc410263937)

**[1.3.](#_Toc410263938)****[研究目的](#_Toc410263938)** [- 1 -](#_Toc410263938)

[**1.4.** **研究方法** - 1 -](#_Toc410263939)

[**1.5.** **プロジェクトマネジメントとの関連性** - 2 -](#_Toc410263940)

[**第2章** **ソフトウェア開発について**](#_Toc410263941)

[**2.1.** **本章の構成** - 2 -](#_Toc410263942)

[**2.2.** **チケットの解説** - 2 -](#_Toc410263943)

[**2.2.1.** **チケットの登録** - 5 -](#_Toc410263944)

[**2.2.2.** **チケットの一覧表示** - 6 -](#_Toc410263945)

[**2.2.3.** **チケットの項目** - 7 -](#_Toc410263946)

[**2.3.** **チケット駆動開発の解説** - 8 -](#_Toc410263947)

[**2.3.1.** **チケット駆動開発の開発サイクル** - 10 -](#_Toc410263948)

[**2.3.2.** **チケット駆動開発のメリット** - 10 -](#_Toc410263949)

[**2.3.3.** **チケット駆動開発のデメリット** - 10 -](#_Toc410263950)

[**第3章** **Gitについて**](#_Toc410263951)

[**3.1.** **本章の構成** - 11 -](#_Toc410263952)

[**3.2.** **GitHubの解説** - 11 -](#_Toc410263953)

[**3.3.** **GitHubの用語** - 11 -](#_Toc410263954)

[**3.4.** **バージョン管理システムの解説** - 13 -](#_Toc410263955)

[**3.4.1.** **集中管理方式「Subversion (SVN) 」** - 13 -](#_Toc410263956)

[**3.4.2.** **分散管理方式「Git」** - 14 -](#_Toc410263957)

[**3.5.** **Gitの解説** - 15 -](#_Toc410263958)

[**3.5.1.** **Gitコマンド** - 15 -](#_Toc410263959)

[**3.6.** **API (Application Programming Interface) の解説** - 31 -](#_Toc410263960)

[**3.7.** **GitHub APIのコード** - 31 -](#_Toc410263961)

[**第4章** **開発・調査**](#_Toc410263962)

[**4.1.** **本章の構成** - 33 -](#_Toc410263963)

[**4.2.** **調査対象** - 33 -](#_Toc410263964)

[**4.2.1.** **調査対象データ** - 33 -](#_Toc410263965)

[**4.2.2.** **調査対象プロジェクト** - 35 -](#_Toc410263966)

[**4.3.** **調査環境の構築** - 37 -](#_Toc410263967)

[**4.3.1.** **Ubuntuの解説** - 37 -](#_Toc410263968)

[**4.3.2.** **調査環境の構築と処理テスト** - 38 -](#_Toc410263969)

[**4.4.** **調査方法** - 41 -](#_Toc410263970)

[**4.4.1.** **プロジェクト成功の成否の調査** - 41 -](#_Toc410263971)

[**4.4.2.** **プロジェクト開始時期の調査** - 44 -](#_Toc410263972)

[**4.4.3.** **Issuesの調査** - 45 -](#_Toc410263973)

[**4.4.4.** **Star数，Contributors数，Fork数の調査** - 49 -](#_Toc410263974)

[**4.4.5.** **プロジェクトの分類** - 50 -](#_Toc410263975)

[**4.4.6.** **重回帰分析** - 59 -](#_Toc410263976)

[**4.4.7.** **プログラムの開発** - 63 -](#_Toc410263977)

[**第5章** **結果・考察**](#_Toc410263978)

[**5.1.** **本章の構成** - 65 -](#_Toc410263979)

[**5.2.** **調査結果** - 65 -](#_Toc410263980)

[**5.2.1.** **プロジェクト成功の成否の調査結果** - 65 -](#_Toc410263981)

[**5.2.2.** **Star数ランキング上位50件のプロジェクトの調査結果** - 66 -](#_Toc410263982)

[**5.2.3.** **ランダムに選択した50件のプロジェクトの調査結果** - 75 -](#_Toc410263983)

[**5.3.** **調査結果まとめ** - 84 -](#_Toc410263984)

[**5.4.** **考察** - 85 -](#_Toc410263985)

[**参考文献**](#_Toc410263986)

**図目次**

[図 1　チケットの処理フロー - 3 -](#_Toc410263987)

[図 2　チケットの登録画面 - 5 -](#_Toc410263988)

[図 3　チケットの一覧表示画面 - 6 -](#_Toc410263989)

[図 4　集中管理方式 - 13 -](#_Toc410263990)

[図 5　分散管理方式 - 14 -](#_Toc410263991)

[図 6　git commit –amend - 26 -](#_Toc410263992)

[図 7　git rebase - 27 -](#_Toc410263993)

[図 8　Star数の調査 - 41 -](#_Toc410263994)

[図 9　リリースバージョンの調査 - 42 -](#_Toc410263995)

[図 10　Commits履歴の調査結果 - 44 -](#_Toc410263996)

[図 11　線形近似曲線の式 - 47 -](#_Toc410263997)

[図 12　Issuesの調査結果 - 48 -](#_Toc410263998)

[図 13　 Star数，Contributors数，Fork数の調査結果 - 49 -](#_Toc410263999)

[図 14　RankData.csv - 50 -](#_Toc410264000)

[図 15　Data1の表示 - 53 -](#_Toc410264001)

[図 16　階層クラスター分析結果 - 54 -](#_Toc410264002)

[図 17　非階層クラスター分析結果 - 55 -](#_Toc410264003)

[図 18　ポジショニングマップ - 57 -](#_Toc410264004)

[図 19　各ユニットのオブザベーション数 - 57 -](#_Toc410264005)

[図 20　コード情報 - 58 -](#_Toc410264006)

[図 21　類似度の変化 - 59 -](#_Toc410264007)

[図 22　重回帰分析結果 - 60 -](#_Toc410264008)

[図 23　変数選択後の重回帰分析結果 - 61 -](#_Toc410264009)

[図 24　MyModel2の相関関係図 - 62 -](#_Toc410264010)

[図 25　階層クラスター分析結果　上位50件 - 66 -](#_Toc410264011)

[図 26　クラスターラベル1　上位50件 - 67 -](#_Toc410264012)

[図 27　クラスターラベル2　上位50件 - 67 -](#_Toc410264013)

[図 28　クラスターラベル3　上位50件 - 68 -](#_Toc410264014)

[図 29　クラスターラベル4　上位50件 - 68 -](#_Toc410264015)

[図 30　非階層クラスター分析結果　上位50件 - 69 -](#_Toc410264016)

[図 31　ポジショニングマップ　上位50件 - 70 -](#_Toc410264017)

[図 32　各ユニットのオブザベーション数　上位50件 - 71 -](#_Toc410264018)

[図 33　コード情報　上位50件 - 71 -](#_Toc410264019)

[図 34　類似度の変化　上位50件 - 72 -](#_Toc410264020)

[図 35　重回帰分析結果　上位50件 - 73 -](#_Toc410264021)

[図 36　相関関係図　上位50件 - 74 -](#_Toc410264022)

[図 37　階層クラスター分析結果　ランダム50件 - 75 -](#_Toc410264023)

[図 38　クラスターラベル1　ランダム50件 - 76 -](#_Toc410264024)

[図 39　クラスターラベル2　ランダム50件 - 77 -](#_Toc410264025)

[図 40　クラスターラベル3　ランダム50件 - 77 -](#_Toc410264026)

[図 41　クラスターラベル4　ランダム50件 - 78 -](#_Toc410264027)

[図 42　クラスターラベル5　ランダム50件 - 78 -](#_Toc410264028)

[図 43　非階層クラスター分析結果　ランダム50件 - 79 -](#_Toc410264029)

[図 44　ポジショニングマップ　ランダム50件 - 80 -](#_Toc410264030)

[図 45　各ユニットのオブザベーション数　ランダム50件 - 81 -](#_Toc410264031)

[図 46　コード情報　ランダム50件 - 81 -](#_Toc410264032)

[図 47　類似度の変化　ランダム50件 - 82 -](#_Toc410264033)

[図 48　重回帰分析　ランダム50件 - 83 -](#_Toc410264034)

[図 49　相関関係図　ランダム50件 - 84 -](#_Toc410264035)

**表目次**

[表 1　チケットの項目 - 7 -](#_Toc410264049)

[表 2　GitHubの用語一覧 - 11 -](#_Toc410264050)

[表 3　調査対象データ - 34 -](#_Toc410264051)

[表 4　調査対象プロジェクト　上位50件 - 35 -](#_Toc410264052)

[表 5　調査対象プロジェクト　ランダム50件 - 36 -](#_Toc410264053)

[表 6　Star数ランキング上位50件のプロジェクト (Rank.csv) - 43 -](#_Toc410264054)

[表 7　ランダムに選択した50件のプロジェクト (Random.csv) - 43 -](#_Toc410264055)

[表 8　Issuesの累積数と経過日数 - 46 -](#_Toc410264056)

[表 9　分析に使用する変数 - 51 -](#_Toc410264057)

[表 10　 HClust.1のサマリ - 54 -](#_Toc410264058)

[表 11 　HClust.1の主成分表 - 55 -](#_Toc410264059)

[表 12　Star数ランキング上位50件のプロジェクトの成否表 - 65 -](#_Toc410264060)

[表 13　ランダムに選択した50件のプロジェクトの成否表 - 65 -](#_Toc410264061)

第1章

序論

# **序論**

* 1. **本章の構成**

本章では，本研究の背景・目的・方法・プロジェクトマネジメントとの関連性を記す．

* 1. **研究背景**

ソフトウェア開発のためのホスティングサービスであるGitHubでは様々なソフトウェアが開発されている．2013年12月にはGitHub上に1000万件のリポジトリが作成され，ユーザ数は400万人を超えた．数多くのプロジェクトが公開されているGitHubを調査すれば，ソフトウェア開発プロジェクトの分類が可能であると考えられる．

　過去にGitHub上のプロジェクトのチケットを調査し，プロジェクトを分類するという研究があり，プロジェクトの分類が可能であるということが明らかにされていた[2]．しかし，この研究では分類の解釈を人間が主観的に行っており，客観性に欠けているという問題があった．そのため，本研究ではデータマイニング手法を用いて分類を客観的に行う．

　GitHubには，リポジトリの人気指標の1つにStarが存在する．Starとは，気になるリポジトリをブックマークできる機能である．他者からStarを押された数をStar数とし，Star数が多いリポジトリは人気が高いことを示している．本研究では，Star数を基準にプロジェクトを選択し，調査をする．

　本研究では，プロジェクトを分類するためにチケットを調査する．チケットとは，ソフトウェア開発中に発生した作業や変更履歴の内容を登録する作業指示書である．チケットには未完了チケットと完了済チケットの2種類が存在する．未完了チケットは作業が完了されていないチケットを示し，完了済チケットは作業が完了されているチケットを示す．チケットによって作業の進捗状況を可視化できるため，進捗管理が容易になる．

　このチケットを中心に開発する手法をチケット駆動開発という．これは作業を開始する前に必ずチケットを発行することを原則とした開発手法である．この開発手法を運用しているプロジェクトは，未完了チケット数と完了済チケット数の時系列変化から進捗状況を判断できる．

* 1. **研究目的**

　GitHub上のプロジェクトを対象とし，プロジェクト成功の成否を調査する．調査後，チケット数の時系列変化に着目し，データマイニング手法を用いて成功の成否に関連するパターンを発見する．

* 1. **研究方法**

本研究では，Star数がプロジェクトの成功要因であると仮定する．この仮定の真偽を検証するため，100件のプロジェクトからStar数とリリースされたソフトウェアのバージョンを取得し，相関関係を調査する．成功の成否は，リリースされたソフトウェアのバージョンから判断する．Ver. 1.0以上をリリースしているプロジェクトを成功とし，Ver. 1.0未満でリリースが止まっているプロジェクトを失敗とする．

プロジェクトの分類にはIssuesを利用する．Issues（GitHub上でのチケット）はGitHub内のStar数ランキング上位50件のプロジェクトとランダムに選択した50件のプロジェクトから，APIを用いて取得する．取得するIssuesはOpenIssues（GitHub上での未完了チケット）とClosedIssues（GitHub上での完了済チケット）の2種類である．

分類に使用する解析手法は，階層クラスター分析と非階層クラスター分析，自己組織化マップである．その変数は，時系列データにフィットする線形式の係数とIssuesの数，Issuesを完了するまでの所要時間の平均と標準偏差である．分類結果から，成功の成否に関連するパターンを発見する．

* 1. **プロジェクトマネジメントとの関連性**

　チケットは，ソフトウェア開発中に発生した課題やバグの内容を可視化し，進捗管理を容易にできる．これはPMBOKが提唱するスコープマネジメントに関連する．プロジェクトマネジメントにおいて進捗管理は重要であり，この作業を効率化できるチケットを調査することは有用であると考える．

第2章

チケットについて

# **ソフトウェア開発について**

* 1. **本章の構成**

本章では，本研究で調査するチケットの基本知識と使用方法について記述する．また，チケットを中心とした開発手法であるチケット駆動開発の基本知識と使用方法についても記す．

* 1. **チケットの解説**

チケットとは，課題管理システム (Issue Tracking System ; ITS) やバグ管理システム (Bug Tracking System ; BTS) で使用される進捗管理ツールである．1つの課題やバグを管理する単位をチケットと呼び，ソフトウェア開発中に発生した課題やバグの内容を記述する．チケットには，未完了チケットと完了済チケットの2種類が存在する．未完了チケットは作業が完了されていないチケットを示し，完了済チケットは作業が完了されているチケットを示す．以下にチケットの処理フローを記載する．

* 検証担当者

アプリケーションの検証を行い，バグを発見した際にチケットを作成します．また，修正結果に対しての再検証を行う．

* 修正担当者

バグの詳細が記述されたチケットを受け，アプリケーションの修正を行う．

* 管理者

担当者のアサイン，検証結果の承認を行う．

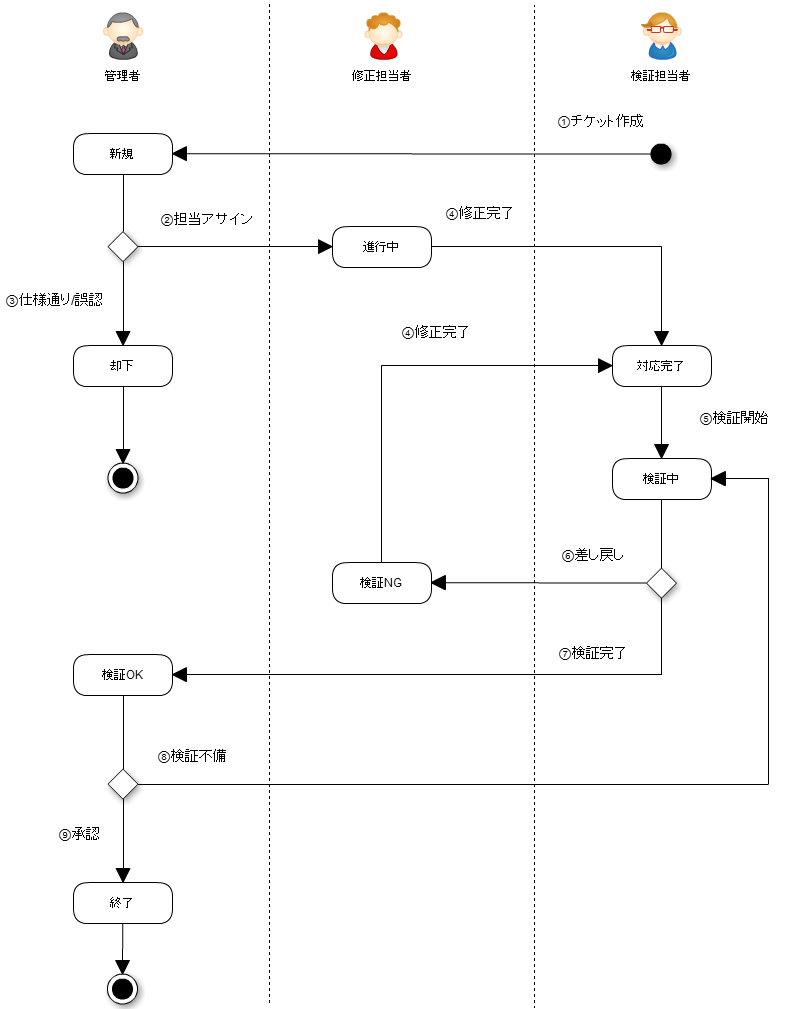


図 1　チケットの処理フロー

1. チケット作成［→ 新規］

検証担当者はバグを発見すると，バグの詳細を記述したチケットを新規に作成します．作成の際に担当者を管理者にすることで，バグが発見された通知が管理者に送られます．

1. 担当アサイン［新規 → 進行中］

管理者はチケットに記述されたバグ情報を確認し，修正担当者をアサインします．

1. 仕様通り／誤認［新規 → 却下］

　チケットに記述されたバグ情報を確認し，そもそも仕様通りの場合や検証担当者の誤認だった場合は対応不要のため，理由を記述したうえでチケットのステータスを"却下"に変更し，終了します．

1. 修正完了［進行中 → 対応完了］［検証NG → 対応完了］

修正担当者はバグの修正対応を行い，その対応の詳細をチケットに記述します．再度検証を行うため，担当者をテスト担当者に変更します．

1. 検証開始［対応完了 → 検証中］

対応完了になっているチケットから，検証を開始するチケットを選択し「検証中」に変更します．

1. 差し戻し［検証中 → 検証NG］

バグの修正対応が不十分であった場合，「検証NG」として差し戻しを行います．

1. 検証完了［検証中 → 検証OK］

バグの修正対応に問題がないと確認できた場合「検証OK」とし，管理者に最終承認を行ってもらいます．

1. 検証不備［検証OK → 検証中］

検証結果を確認し，検証内容に不備があった場合や，追加で検証を行ってほしい項目などがあれば， 検証担当者に追加の検証を行ってもらいます．

1. 承認［検証OK → 終了］

検証結果に問題がなければ承認とし，チケットのステータスを「終了」にします[6]．

* + 1. **チケットの登録**

チケットは，ソフトウェア開発中に課題やバグが発生した場合に登録される．チケットの登録はプロジェクトのメンバが行え，登録されたチケットはメンバ間で閲覧できる．チケットの項目には，題名・作業内容・ステータス・優先度・担当者・期日などがあり，チケットを登録する際に記述できる．また，Labelを設定することで，チケットがどのような種類であるのかを指定できる．以下の図は，チケットと同じ役割を持つGitHubのIssuesである．

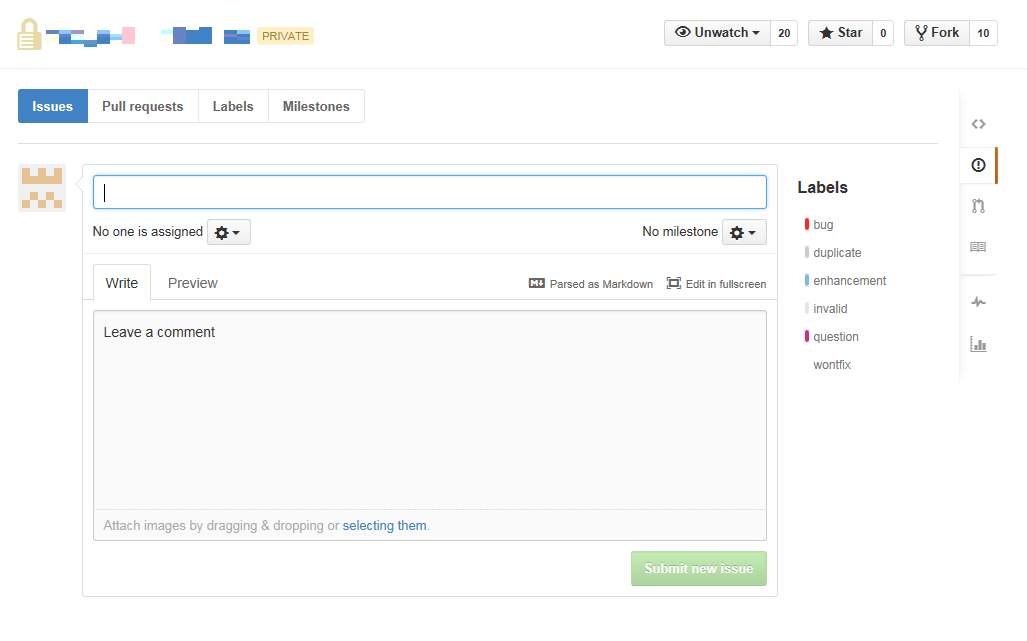


図 2　チケットの登録画面

* + 1. **チケットの一覧表示**

登録されたチケットは，検索機能やソート機能によって検索・並び替え・一覧表示が可能である．例として，Labelを利用した種類ごとの分類表示やキーワード検索，優先度順に並び替えるなどが可能である．一覧表示では，OpenとClosedを別々に表示できるため，作業の進捗状況を把握できる．以下の図は，チケットと同じ役割を持つGitHubのIssuesの一覧表示画面である．

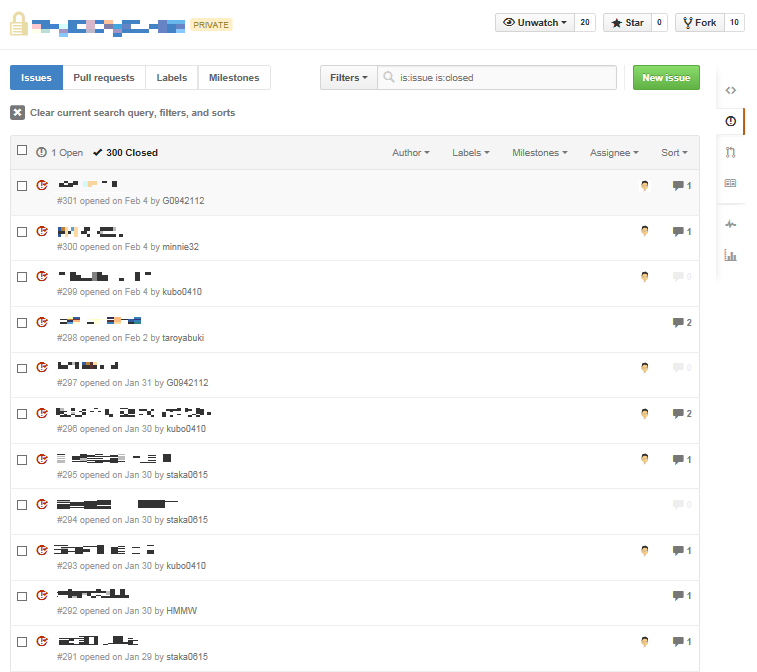


図 3　チケットの一覧表示画面

* + 1. **チケットの項目**

チケットには様々な項目が存在し，登録する際に記述できる．これらの項目により，チケットがどのような理由から登録されたのかを把握できる．以下の表は，チケットの項目である．

表 1　チケットの項目

|  |  |
| --- | --- |
| 項目名 | 解説 |
| Label | チケットがどのような種類であるのかを表す．種類には，バグ・重複・機能追加・質問・対応無しなどがある． |
| 題名 | チケットの一覧表示画面などに表示される．チケットの内容を端的に表すものを記述する． |
| 作業内容 | 題名だけでは書ききれない詳細な作業内容を記述する． |
| ステータス | チケットの進捗状況を表し，Openであるか，Closedであるかを記述する． |
| 優先度 | どのチケットから優先的に着手すべきかを明示できる．チケットの優先度を記述する． |
| 登録者 | チケットを登録したメンバの名前を記述する． |
| 担当者 | チケットを処理すべき担当者の名前を記述する． |
| チケット番号 | チケットが登録された際に割り振られる番号である．何番目に登録されたチケットなのかを表す． |
| 対象バージョン | 登録されたチケットをどのバージョンに関連付けるのかを指定する． |
| 進捗率 | 作業の進捗率を記述する． |
| 開始日 | 作業を開始すべき日付を記述する． |
| 期日 | 作業の期日を記述する． |

* 1. **チケット駆動開発の解説**

　チケット駆動開発 (Ticket-Driven Development ; TiDD) とは，チケット管理から生まれたプロジェクト管理技法の1つである．ITS/BTSのチケットを用いて作業を管理し，作業を開始する前に必ずチケットの登録をすることを原則とした開発手法である．作業管理にチケットを用いることで，作業漏れの防止や頻繁な作業の変更に対応できるようになる．チケット駆動開発には，以下の様な規律が存在する．

* プラクティス1．チケット無しのコミット不可 (No Ticket, No Commit)

プログラムなどの成果物を変更する場合，必ずチケットに変更履歴を残します．バージョン管理のコミットフック機能 (post-commit-hook) を利用して，プログラムをコミットする時，必ずコミットログにチケット番号を記録する運用から生まれた基本ルールです．作業の変更管理を強化し，プログラムの変更履歴をチケット経由で追跡できる利点 (traceability) があります．また，開発者は，1 日の作業の開始前にチケットを登録 (No Ticket, No Work) して，チケットに基づいてプログラム修正を行い，プログラムのコミットと同時にチケットを完了するという開発のリズムが生まれます．

* プラクティス2．チケット無しの作業不可 (No Ticket, No Work)

作業を開始する前に，必ずチケットに作業内容を登録してから作業を開始します．そして，チケットは仕様書ではなく，作業指示書になります．作業履歴がチケットに必ず残るため，他の開発者が参照できたり，収集したチケットから予定・実績の工数集計，進捗・障害の集計結果が得られるため，原因分析や是正対策作りに役立てることができます．

* プラクティス3．イテレーションはリリースバージョンである (Iteration is Version)

イテレーションはソフトウェアのリリースバージョン，プロジェクトのマイルストーンと一致させます．XP のイテレーション (Scrum のスプリント) をバージョン管理の配下にあるブランチのリリースバージョンと同一視することで，イテレーション終了時には必ずソフトウェアをリリースできるというリリース完了条件が導かれます．また，開発チームは，イテレーション単位に定期的に小規模リリースしていくことによって，システムを漸進的に開発していくリズムが生まれます．

* プラクティス4．チケットは製品に従う

ITS プロジェクトというチケット管理の集合は，リリースされる製品の構成 (アーキテクチャ) に従属するように対応付けます．機能追加や障害修正のチケットはソースに対して管理する (No Ticket, No Commit) ため，製品の構成に基づくバージョン管理リポジトリと対応付けるように，ITS プロジェクトを作る方が管理が楽になります．そして，Conway’s Law (組織はアーキテクチャに従う) に従えば，開発組織はチケット管理の階層構造に組み込まれて，チケット駆動で開発しやすい組織体制の変化が促されます．

パッケージ製品を顧客ごとにカスタマイズする派生開発や，複数の製品系列の開発であるソフトウェアプロダクトライン (SPLE) にも適用できます．

* プラクティス5．タスクはチケットで分割統治 (Divide and Conquer)

チケットの粒度が大きい場合，タスクの分割という観点から，開発者が作業しやすいようにチケットを分割します．チケット駆動で運用するといつも迷ってしまうのは，チケットの粒度です．基本的な運用方針としてはチケットの粒度は 1 ～ 5 人日程度まで細分化します．なぜならば，1 日の開発リズムが生まれやすいため，開発者は作業しやすいからです．従って，チケットの作業内容が大きすぎると気づいたら，たとえチケットが登録された後でも，作業しやすい粒度までチケットを分割します．

* プラクティス6．チケットの棚卸し

定期的に，放置された未完了チケットを精査して最新化します．チケット駆動を中心にプロジェクトを運営すると，チケットはどんどん登録され更新されるため，何らかの規律がなければ，乱発・放置されるチケットの重みで開発速度が遅くなってしまいます．その状況を打開するために必要な作業である「チケットの棚卸し」には以下の 4 つの注意点があります．

1. 役割分担

開発チームのリーダーがチケット管理の最終責任者であると認識し，リーダーは定期的に，誰も手を付けない作業やチームの開発の支障となる課題を優先順位付けしたり，ステータスを最新化したりします．

1. 棚卸しのタイミング

例えば，毎日の朝会やリリース後のふりかえりミーティングのように，棚卸しのタイミングを故意に作ります．

1. チームの開発速度 (Velocity) を超えるチケットは後回し

現状のメンバーの技術力やチームの成熟度の観点では，どう考えても実施不可能なほど大量のチケットであふれている状況があります．その場合，チームの開発速度 (Velocity) を超えるチケットはイテレーションから削減し，チームが消化できるレベルのチケットの枚数になるように調整します．

1. 作業不要のチケットは，リリース未定の特別なイテレーションへ移す

作業不要のチケットは，別のイテレーションへ延期したり，リリース期限が未定の特別なイテレーションへ移動して除去します．この特別なイテレーションは，「Unplanned」「バックログ」「Icebox」などとも呼ばれ，次のイテレーション計画作成中に必要なチケットがあると判断されれば，そのチケットを取り込む運用になります．

* プラクティス7．ペア作業 (Pair Work)

一つのチケットを二人以上で連携する作業です．XP のペアプログラミングでは必ず二人が同時間帯に一つの机で作業しますが，ペア作業では一つのチケットの作業結果を受けて，チケットのステータスを更新しながら非同期にペアで作業します．例えば，障害修正において開発者とテスト担当者が交互に修正と検証を行ったり，コードレビューをレビューアと開発者 (レビューイ) が交互にレビューと指摘事項の反映を行ったりする時に使われます．一つのチケットを二人以上の目を通してチェックし，成果物の品質を向上できる利点があります[7]．

* + 1. **チケット駆動開発の開発サイクル**

チケット駆動開発では，概ね次のようなPDCAサイクルを繰り返して開発が行われる．

1. 大まかなリリース計画を作る．
2. 仕事を細かいタスクに分割し，タスクを書き出す (チケットの発行) ．
3. イテレーション単位でタスクをまとめて，イテレーション計画を作る．
4. タスクを一つ選び，実装する．
5. 差分をコミットし，完了する (チケットのクローズ) ．
6. イテレーションに紐づくタスクがすべて終了ステータスになるとリリースする．
7. リリース後，開発チームで作業をふりかえる．
8. 次のイテレーション計画へ顧客の要望やふりかえりの内容を反映する[8]．
   * 1. **チケット駆動開発のメリット**

* 作業管理が容易

作業の全てをチケット経由で行うため，作業状況を可視化でき，明確に作業管理ができる．これにより，作業漏れや頻繁な作業の変更に対応できる．また，各担当者がメンバの作業を閲覧できるため，メンバ間での情報共有が容易に行える．

* 進捗管理が容易

チケットに進捗率や期日を記述できるので，作業の進捗状況を把握しやすい．また，優先度を記述することにより，優先的に着手しなければならない作業を明示できる．

* 変更履歴の管理が可能

ドキュメントやソースコードの変更履歴をチケットに連動できるため，変更履歴を管理できる．これにより，変更後のデータに不具合が見つかった場合でも，履歴をたどることで変更前のデータに戻すことができる．

* マネジメントが容易

作業を割り振られているメンバごとにチケットを閲覧できるので，リソースの空き状況を把握しやすい．

* + 1. **チケット駆動開発のデメリット**
* チケットの乱立

登録されたチケットが完了されずに未完了チケットばかりが増加すると，管理するチケット数が膨大な量になり，マネジメントや進捗管理に支障をきたす．

* チケットの登録作業

作業を開始する前に必ずチケットを登録しなければならないため，進捗状況を文章化する作業が増える．

* チケット駆動開発の理解

チケット駆動開発に関しての知識がない場合，開発作業とは別にチケットの登録方法などを学習する必要がある．また，混乱を防ぐためにもチケットの粒度や必須項目についてメンバ間でルールを決める必要がある．

第3章

GitHubについて

# **Gitについて**

* 1. **本章の構成**

本章では，GitHubとGitの基本知識，使用方法について記す．また，GitHubのAPIに関しても記す．

* 1. **GitHubの解説**

バージョン管理システムGitのプロジェクトをインターネット上で共有・公開することができるネットサービスの一つ．同名の企業 (GitHub社) が運営している．オープンソースソフトウェアの開発プロジェクトなどでソースコードの管理や公開によく用いられ，最も人気の高いGitホスティングサービスの一つである．

Gitで保管・管理するデータの集積 (リポジトリ) をGitHubの運用するサーバ上に集積し，組織内や複数人で共有したり，広く一般に公開したりすることができる．リポジトリはGitを用いて操作できるほか，Webサイト上にも情報が公開され，Webブラウザを通じて閲覧や操作ができる．サイト上では利用者 (開発者) 間のコミュニケーションが可能で，一種のSNSとしても機能している．

パブリックリポジトリ (一般に公開されるリポジトリ) は無料で作成することができるが，プライベートリポジトリ (企業内プロジェクトなどで利用するための非公開のリポジトリ) の作成・利用には月額料金がかかる[9]．

* 1. **GitHubの用語**

GitHubには開発を効率的に行うための機能が多く存在する．その中で使われる用語を一部抜粋し，以下に記述する．

表 2　GitHubの用語一覧

|  |  |
| --- | --- |
| 用語名 | 解説 |
| Repository  (リポジトリ) | アプリケーションやシステム情報などのデータが保持されている貯蔵庫のようなもの．ファイルやディレクトリなどをオブジェクトとして表現する． |
| Clone  (クローン) | サーバに存在するリモートリポジトリを手元のPC (ローカル)にコピーする． |
| Fork  (フォーク) | GitHub上で公開されているリポジトリを自分のリポジトリとして複製する． |
| Origin  (オリジン) | Clone元のリモートリポジトリを示す． |
| Current directory  (作業ディレクトリ) | ユーザが作業しているディレクトリを示す． |

|  |  |
| --- | --- |
| Index  (ステージング)  (ステージングエリア) | ステージングとは，変更したデータのうち，コミットする対象を選別する作業である．ステージングエリアに登録することを「ステージングする」と呼ぶ．  ステージングエリアとは，Gitディレクトリに含まれる，次のコミットに関しての情報が保持された1つのファイルである． |
| Git directory  (ギットディレクトリ) | Gitリポジトリの全ての変更履歴を保持しているディレクトリを示す． |
| Revert  (リベート) | ステージングエリアに追加した変更を戻す． |
| Commits  (コミット) | ステージングエリアに追加した変更をGitディレクトリ上に変更履歴として反映する． |
| Branch  (ブランチ) | メインの流れとは異なる流れで作業をする仕組みである．メインの流れでバグ修正や機能改善を行い，異なる流れで新機能の開発を行う． |
| Merge  (マージ) | 異なるブランチを1つのブランチに統合する． |
| Push  (プッシュ) | ローカルリポジトリに追加した変更履歴をリモートリポジトリに反映する． |
| Pull  (プル) | リモートリポジトリに追加された変更履歴をローカルリポジトリに反映する． |
| Pull Request  (プルリクエスト) | Forkしたリポジトリに対して変更を加え，その変更をFork元のリポジトリに反映してもらうようにリクエストを送る作業である． |
| Issues  (イシュー) | 開発中に発生したバグや課題を記述し，登録する機能．1つのバグや課題に1つのIssuesが割り当てられる．プロジェクトメンバ間での共有が可能であり，コメント，削除などの操作ができる．課題が未完了のIssuesをOpenIssuesと呼び，完了済みのIssuesをClosedIssuesと呼ぶ．チケット駆動開発のチケットと同様な役割を持つ． |
| Star  (スター) | ソーシャルブックマークと同様な機能であり，スターを付けたリポジトリを即座に閲覧できるようになる． |
| Watch  (ウォッチ) | WatchしたリポジトリへのpushやIssuesの作成，Pull Requestの作成などの情報を閲覧できるようになる． |
| Contributors  (コントリビュータ) | GitHub上のプロジェクトへの寄与貢献者．Pull Requestの承認やリポジトリの各種操作を行える権限を持つ． |
| Releases  (リリース) | GitHub上で開発したソフトウェアをGitHub上で配布できる機能である． |

* 1. **バージョン管理システムの解説**

バージョン管理システムとは，ファイルに対して「誰が」「いつ」「何を変更したか」というような情報を記録することで，過去のある時点の状態を復元したり変更内容の差分を表示できるようにするシステムのことです．バージョン管理システムは大きく2つに分けると，「集中管理方式」「分散管理方式」があります．

過去には集中管理方式の「CVS」「Subversion」が多く利用されていましたが，複数人での分散開発の容易さやパフォーマンスに優れた分散管理方式の「Git」「Mercurial」などがスタンダードになりつつあります[10]．

* + 1. **集中管理方式「Subversion (SVN) 」**

　SVNはGitが登場する前から使われている集中型のバージョン管理システムです．SVNよりも先に公開されたバージョン管理システムに「CVS」がありますが，SVNはCVSを参考にして開発されました．まずはSVNについて見ていきます．

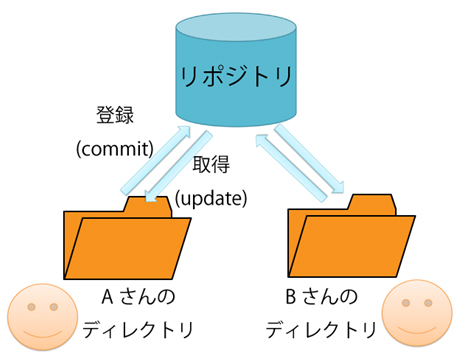


図 4　集中管理方式

　SVNは集中型 (クライアント・サーバ型) のバージョン管理システムです．ファイルそのものや変更の履歴などを保存する場所を「リポジトリ」 (貯蔵庫) と呼びますが，SVNの場合は (ソフトウェア1つにつき) 1つのリポジトリを使います．ソフトウェア開発に参加するメンバーは，中央リポジトリ (プロジェクトメンバー間で共有するリポジトリ) からソースコードを持ってきて編集し，編集が終わったら中央リポジトリに直接反映します．SVNは集中型のバージョン管理システムなので，リポジトリが置かれたサーバに接続できない環境の場合，最新のソースコードを取得やファイル編集の反映ができません[11]．

* + 1. **分散管理方式「Git」**

　集中型バージョン管理システムのSVNに対し，Gitは分散型のバージョン管理システムです．リポジトリを複数持つことができ，開発の形態や規模に合わせてソースコードの管理ができます．リポジトリを複数用意できるので「分散型」と呼ばれています．

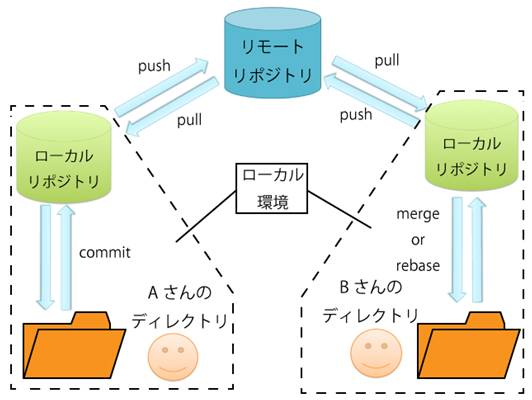


図 5　分散管理方式

　例えば上の図のように，リモートリポジトリをサーバ上に置き，開発者それぞれがローカルにリポジトリを持つという構成が考えられます．この場合，普段はローカルリポジトリを使って作業し，ある程度作業できたらリモートリポジトリに反映するといった使い方ができます．リモートリポジトリにアクセスできない環境でも作業を進めることができます．また，大きな規模のソフトウェアの場合，個人や小さなチームで試験的に実装を進めて，ある程度進んだ時点で親のリポジトリや他の開発者のリポジトリに反映するといった使い方ができます．開発者同士の作業を柔軟に進めることができ，オープンソースのプロジェクトなどにも適しています[11]．

* 1. **Gitの解説**

オープンソースの分散バージョン管理システムの一つ．複数の開発者が共同で一つのソフトウェアを開発する際などに，ソースコードやドキュメントなどの編集履歴を管理するのによく用いられる．

バージョン管理システムの基本的な機能として，個々のファイルにいつ誰がどのような変更を行ったかを記録しており，必要に応じて特定の日時の版を参照したり，その状態に戻したりすることができる．また，プロジェクトの時系列を分岐 (ブランチ) して派生プロジェクトを作成したり，それを再び元の系列に融合 (マージ) したりすることができる．

Gitの特徴として，管理するデータの集合体であるリポジトリを利用者の手元のコンピュータに丸ごとコピーし，必要に応じて中央の管理サーバと同期する方式を取っている．これにより，常にサーバと通信可能でなくても編集が可能で，検索や参照なども高速に行なうことができる．

　GitHubのようにインターネット上のサーバにGitの中央リポジトリを作成・運用できるサービスもあり，GitHubに作成したリポジトリに各開発者がGitでアクセスして開発を進めるというスタイルがオープンソースソフトウェアなどで人気となっている[12]．

* + 1. **Gitコマンド**

　GitHubはGitホスティングサービスであるため，リポジトリの作成・運用をGitコマンドで行う．以下に，リポジトリを操作するうえで重要であるGitコマンドを記載する．

* git init

git init はGitリポジトリを新たに作成するコマンドです．このコマンドは，バージョン管理を行っていない既存のプロジェクトをGitリポジトリに変換する場合や，空の新規リポジトリを作成して初期化する場合に使用します．このコマンドを除く他のコマンドはほとんどすべて初期化されたリポジトリ以外には適用することができないため，このコマンドは新規プロジェクトを開始する場合に通常最初に実行するコマンドです．

git init コマンドを実行すると，リポジトリに付随するすべてのメタデータを有する .git サブディレクトリがプロジェクトのルートに作成されます．この .git ディレクトリを除けば，既存プロジェクトには何の改変も行われません (SVNとは異なり，Gitでは各々のサブディレクトリに .gitフォルダーが作成されることはありません) ．

|  |
| --- |
| git init |

　カレントディレクトリを Gitリポジトリに変換します．このコマンドを実行するとカレントディレクトリに .gitフォルダーが作成され，プロジェクトのバージョンの管理を開始することができます．

|  |
| --- |
| git init <directory> |

　指定したディレクトリに空の Gitリポジトリを作成します．このコマンドを実行すると，.gitサブディレクトリのみを含む <directory> という名称の新規フォルダーが作成されます．

|  |
| --- |
| git init --bare <directory> |

　作業ディレクトリを持たない空のGitリポジトリを作成して初期化します．共有リポジトリは必ず--bareフラグを指定して作成しなければなりません (下の補足説明参照) ．--bareフラグを指定して作成したリポジトリの名称の最後には習慣的に.gitを付加します．例えば，my-projectという名称のリポジトリのベアバージョンは，my-project.gitという名称のフォルダーに格納します．

* git clone

git cloneは，既存のGit リポジトリのコピーを作成するコマンドです．このコマンドはsvn checkoutと似ていますが，作業コピーがそれ自身で完全なGitリポジトリを構成する点が異なっていて，即ち作業コピーは自分自身の履歴を持ち，自分自身でファイルを管理し，元のリポジトリとは完全に独立した環境を提供します．

利用者の便宜のため，クローンを行うと，元のリポジトリをポイントするoriginという名称のリモート接続を自動的に作成します．これにより，極めて簡単に中央リポジトリとの通信を行うことができます．

|  |
| --- |
| git clone <repo> |

<repo> にあるリポジトリをローカルマシーンにクローンするコマンドです．元のリポジトリはローカルマシーンに存在してもHTTPやSSHを用いてアクセスするリモートマシーンに存在しても構いません．

|  |
| --- |
| git clone <repo> <directory> |

<repo> にあるリポジトリをローカルマシーン上の <directory> という名称のフォルダーにクローンするコマンドです．

* git config

git configは，インストールしたGit (または個々のリポジトリ) に対してコマンドラインから設定を行うコマンドです．このコマンドを使用すると，ユーザー情報からリポジトリ動作の初期設定にいたるまであらゆる項目の設定が可能です．通常使用される設定オプションのいくつかを以下に示します．

|  |
| --- |
| git config user.name <name> |

　現在のリポジトリにおけるすべてのコミットに使われるオーサー名を設定します．このコマンドでは通常，--globalフラグを指定して現在のユーザーを対象としたオーサー名設定を行ないます．

|  |
| --- |
| git config --global user.name <name> |

　現在のユーザーが行うすべてのコミットのオーサー名を設定します．

|  |
| --- |
| git config --global user.email <email> |

　現在のユーザーが行うすべてのコミットに関してそのオーサーEメールアドレスを設定します．

|  |
| --- |
| git config --global alias. <alias-name> <git-command> |

　Git コマンドのショートカットを作成します．

|  |
| --- |
| git config --system core.editor <editor> |

　現在使用しているマシーンにおいてgit commitのようなコマンドを実行する際に使用するエディターを指定します．引数 <editor> は，該当のエディターを起動するコマンドです (例えば，vi エディターなど) ．

|  |
| --- |
| git config --global –edit |

　グローバルな設定ファイルをテキストエディターで開くコマンドで，これを使用して手動で設定ファイルを編集することができます．

* git add

git addは，作業ディレクトリ内の変更をステージングエリアに追加するコマンドです．このコマンドは，個々のファイルのアップデート内容を次回コミットの対象とすることを Git に指示します．ただし，git addコマンドだけでは実際にはローカルリポジトリに何の影響も与えず，git commitコマンドを実行するまでは変更が実際に記録されることはありません．

これらのコマンドと関連して，作業ディレクトリおよびステージングエリアの状態を確認するために，git statusコマンドが用いられます．

|  |
| --- |
| git add <file> |

<file> に加えられたすべての変更をステージして次回のコミットの対象とします．

|  |
| --- |
| git add <directory> |

<directory> 内のすべての変更をステージして次回のコミットの対象とします．

|  |
| --- |
| git add –p |

インタラクティブなステージングセッションを開始します．インタラクティブなステージングセッションでは，ファイルの一部を選択してステージし，次回のコミットの対象とすることができます．このコマンドを実行すると，変更部分が表示され，それに対する次のコマンド入力を要求されます．y を入力するとその部分がステージされ，nを入力するとその部分は無視され，sを入力するとその部分はより小さい部分に分割され，eを入力するとその部分を手作業で編集することが可能となり，qを入力するとインタラクティブなセッションを終了します．

* git commit

git commit は，ステージされたスナップショットをローカルリポジトリにコミットするコマンドです．コミットされたスナップショットはプロジェクトの「安全に保存された」バージョンであると解釈でき，明示的に変更指示が行われない限りGitがそれを変更することはありません．これはgit addと共にGitにおける最も重要な種類のコマンドです．

名称は同じですが，このコマンドはsvn commitコマンドとは全く異なるものです．スナップショットはローカルリポジトリにコミットされるため，他のGitリポジトリには全く影響を与えません．

|  |
| --- |
| git commit |

ステージされたスナップショットをコミットするコマンドです．このコマンドを実行するとテキストエディターが起動され，コミットメッセージの入力を求められます．メッセージの入力後，ファイルを保存してエディターを終了するとコミットが実行されます．

|  |
| --- |
| git commit -m " <message> " |

テキストエディターを起動することなく， <message> をコミットメッセージとして，ステージされたスナップショットをコミットします．

|  |
| --- |
| git commit –a |

作業ディレクトリにおけるすべての変更のスナップショットをコミットします．これには追跡対象ファイル (過去にgit addコマンドによってステージングエリアに追加されたことのあるファイル) の修正のみが含まれます．

* git status

git statusは，作業ディレクトリの状態とステージされたスナップショットの状態を表示するコマンドです．このコマンドを実行すると，ステージされた変更内容，されていない変更内容，Gitによる追跡の対象外となっているファイルが表示されます．このステータス情報出力には，コミット済みの変更履歴に関する情報は含まれません．コミット済みの変更履歴に関する情報を取得する場合は，git logコマンドを使用します．

|  |
| --- |
| git status |

ステージされたファイル，ステージされていないファイル，追跡対象外のファイルを一覧表示します．

* git log

git logは，コミット済みのスナップショットを表示するコマンドです．このコマンドを使用することにより，コミット済み変更履歴の一覧表示，それに対するフィルター処理，特定の変更内容の検索を行うことができます．git statusコマンドは作業ディレクトリとステージングエリアの状態を確認するためのものであるのに対し，git logコマンドはコミット済みの履歴 (コミット履歴) のみが対象です．

|  |
| --- |
| git log |

コミット履歴全体をデフォルトの形式で表示します．出力表示が2ページ以上にわたる場合はSpaceキーでスクロールが可能です．終了する場合はqを入力します．

|  |
| --- |
| git log -n <limit> |

表示するコミット数を <limit> に制限します．例えばgit log -n 3とすると，表示するコミット数は 3 です．

|  |
| --- |
| git log –oneline |

各々のコミットの内容を1行に圧縮して表示するコマンドです．このコマンドは，コミット履歴を概観する目的に適しています．

|  |
| --- |
| git log –stat |

通常のgit log情報に加えて，改変されたファイルおよびその中での追加行数と削除行数を増減数で表示します．

|  |
| --- |
| git log –p |

各々のコミットに対応するパッチを表示します．このコマンドを実行すると，コミット履歴から取得できる最も詳細な情報である各々のコミットの完全な差分情報が表示されます．

|  |
| --- |
| git log --author=" <pattern> " |

特定のオーサーが行なったコミットを検索します．引数 <pattern> にはプレーンテキストまたは正規表現を用いることができます．

|  |
| --- |
| git log --grep=" <pattern> " |

コミットメッセージが <pattern> (プレーンテキストまたは正規表現) と一致するコミットを検索します．

|  |
| --- |
| git log <since> .. <until> |

<since> と <until> の間に位置するコミットのみを表示します．2個の引数には，コミットID，ブランチ名，HEAD，その他任意のリビジョンリファレンスを用いることができます．

|  |
| --- |
| git log <file> |

指定したファイルを含むコミットのみを表示します．このコマンドは，特定のファイルの履歴を調べる目的に便利です．

|  |
| --- |
| git log --graph --decorate ﻿--oneline |

ここにはいくつかの便利なオプションが示されています．--graphフラグを指定すると，コミットメッセージの左側にテキストベースでコミット履歴をグラフ化したもの描画します．--decorateフラグを指定すると，表示されているコミットのブランチ名やタグ名を追加して表示します．--onelineフラグを指定するとコミット情報を圧縮して1行に表示するためコミット履歴の概観に便利です．

* git checkout

git checkoutは，ファイルのチェックアウト，コミットのチェックアウト，ブランチのチェックアウトの3 つの異なる機能を有するコマンドです．この章では最初の二つの機能について説明します．

コミットのチェックアウトを行うと，作業ディレクトリがそのコミットと完全に一致した状態になります．このコマンドは，プロジェクトの現在の状態を一切変更することなく過去の状態を確認する場合に使用します．ファイルのチェックアウトを行うと，作業ディレクトリの他の部分に一切影響を与えることなくそのファイルの過去のリビジョンを確認することができます．

|  |
| --- |
| git checkout master |

masterブランチに戻るコマンドです．ブランチについては次の章で詳しく説明しますが，ここではとりあえずmasterブランチはプロジェクトの「現在の」状態に戻る手段だと考えてください．

|  |
| --- |
| git checkout <commit> <file> |

ファイルの過去のリビジョンをチェックアウトするコマンドです．このコマンドを実行すると，作業ディレクトリに存在する指定した <file> が，指定した <commit> に含まれそのファイルの完全なコピーとなり，さらにそれをステージングエリアに追加します．

|  |
| --- |
| git checkout <commit> |

作業ディレクトリ内のすべてのファイルを指定したコミットと同一の状態に更新するコマンドです．引数 <commit> にはコミットハッシュまたはタグを使用することができます．このコマンドを実行すると，”detached HEAD”状態になります．

* git revert

git revertは，コミットされたスナップショットを元に戻すコマンドです．ただし，プロジェクト履歴においてそのコミットがなかったものとするのではなくそのコミットによって加えられた変更を元に戻す方法を見出してその結果を新しいコミットとして追加するものです．これは，Gitの履歴を保全するためであり，このことはバージョン履歴の完全性の維持とコラボレーションの信頼性の確保には必須です．

|  |
| --- |
| git revert <commit> |

<commit> によって加えられたすべての変更を元に戻す新しいコミットを生成し，それを現在のブランチに適用するコマンドです．

* git reset

　git revertコマンドが変更を元に戻す「安全な」方法であるとするならば，git resetコマンドは「危険な」方法と言うことができます．git resetコマンドを使用して元に戻すと (そしてrefやreflogによるリファレンスが不可能になっている場合) ，元の状態を復元する方法はありません．この「元に戻す」操作を取り消すことは不可能なのです．このコマンドはGitにおいて作業結果を失う可能性のある数少ないコマンドのひとつであり，このコマンドを使用する場合は注意が必要です．

　git checkoutコマンド同様，git resetも様々な設定項目のある応用範囲の広いコマンドです．コミット済みのスナップショットを削除する目的にも使用されますが，ステージングエリアや作業ディレクトリにおける変更を元に戻す場合により多く使われます．いずれにしてもこのコマンドの使用はローカルな変更を元に戻す場合に限るべきであり，他の開発者に公開されているコミットの取り消しは決して行ってはなりません．

|  |
| --- |
| git reset <file> |

　作業ディレクトリに何の変更も加えずに，指定したファイルをステージングエリアから削除するコマンドです．このコマンドを実行すると，変更を書き込むことなく指定したファイルをアンステージします．

|  |
| --- |
| git reset |

　作業ディレクトリに何の変更も加えることなくステージエリアをリセットして直前のコミット時の状態と一致させるコマンドです．このコマンドを実行すると，変更を書き込むことなくすべてのファイルをアンステージし，一度ステージされたスナップショットを初めから再構築することができるようになります．

|  |
| --- |
| git reset --hard |

　ステージエリアと作業ディレクトリをリセットして直前のコミット時の状態と一致させるコマンドです．--hardは，変更をアンステージした上でさらに作業ディレクトリ内のすべての変更を元に戻すことをGitに指示するフラグです．言い換えると，これはコミット前のすべての変更を全くなかったものとするコマンドであり，これを使用する場合はローカルマシーン上で行った開発作業を本当に破棄していいのか否かを確認する必要があります．

|  |
| --- |
| git reset <commit> |

　現在のブランチの先端を <commit> の位置に戻した上でステージングエリアをその状態と一致するように元に戻しますが，作業ディレクトリのみはそのままにしておきます <commit> の実行後に行われた変更は作業ディレクトリに保存されており，より変更規模が小さくて整理されたスナップショットを作成してローカルリポジトリへの再コミットを行うことができます．

|  |
| --- |
| git reset --hard <commit> |

　現在のブランチの先端を <commit> の位置に戻した上でステージングエリアおよび作業ディレクトリをその状態と一致するように元に戻します．このコマンドを実行すると，コミット前の変更に加えて <commit> の後に行われたすべてのコミットも全くなかったものとなります

* git clean

　git cleanは，作業ディレクトリから追跡対象外のファイルを削除するコマンドです．git statusコマンドを使用して追跡対称外のファイルを確認して手作業でそれらを削除することは簡単ですので，このコマンドはどちらかといえば便宜性を高めるために用意されたものです．通常のrmコマンド同様git cleanコマンドも元に戻すことはできないため，このコマンドを実行する際にはその追跡対象外ファイルを本当に削除していいか否かを再確認してください．

　git cleanは，git reset --hardコマンドとよく併用されます．既に説明したようにresetコマンドが作用するのは追跡対象となっているファイルのみであるため，追跡対象外のファイルをクリーンアップするためには別のコマンドが必要になるのです．この2つのコマンドを併用することにより，作業ディレクトリをある特定のコミットの時点と完全に同一の状態に戻すことができます．

|  |
| --- |
| git clean -n |

　git cleanの「予行演習」を行うコマンドです．このコマンドを実行すると削除されるファイルを表示しますが，実際の削除は行われません．

|  |
| --- |
| git clean -f |

　追跡対象外のファイルをカレントディレクトリから削除するコマンドです．設定オプションclean.requireForceがfalseにセットされていない場合(このオプションはデフォルトでは true です)は，-f (force) フラグは必須です．このコマンドでは，追跡対象外のフォルダーやファイルであっても.gitignore.で指定したものは削除しません．

|  |
| --- |
| git clean -f <path> |

　追跡対象外のファイルを削除しますが，その対象範囲は指定したパスに限定するコマンドです．

|  |
| --- |
| git clean -df |

　カレントディレクトリ内の追跡対象外ファイルおよび追跡対象外ディレクトリを削除します．

|  |
| --- |
| git clean -xf |

　カレントディレクトリ内の追跡対象外ファイルおよび Git では通常無視されるファイルを削除します．

* git branch

　ブランチとは独立な開発ラインを意味します．ブランチは，このチュートリアルシリーズの最初の章であるGitの基本 で説明した編集/ステージ/コミットプロセスに対する抽象概念です．これは，作業ディレクトリやステージングエリア，プロジェクト履歴を全く新しく作成する手段であると考えることもできます．新規のコミットは，現在のブランチの履歴に記録され，プロジェクト履歴における分岐を形成します．

　git branchは，ブランチの作成，一覧表示，リネーム，削除を行うコマンドです．このコマンドにはブランチの切り替えを行う機能も，分岐した履歴を統合して元に戻す機能もありません．このため，git branchコマンドは多くの場合git checkoutコマンドおよびgit mergeコマンドと併用されます．

|  |
| --- |
| git branch |

　リポジトリ内のすべてのブランチを一覧表示します．

|  |
| --- |
| git branch <branch> |

<branch> という名称の新規ブランチを作成します．新たに作成されたブラントのチェックアウトは行われません．

|  |
| --- |
| git branch -d <branch> |

　指定したブランチを削除します．ブランチにマージされていない変更が残っている場合はGitが削除を拒否するため，このコマンドは「安全な」操作です．

|  |
| --- |
| git branch -D <branch> |

　指定したブランチにマージされていない変更が残っていたとしてもそれを強制的に削除するコマンドです．このコマンドは，特定の開発ラインで行われたすべてのコミットを完全に破棄する場合に使用します．

|  |
| --- |
| git branch -m <branch> |

　現在のブランチの名前を <branch> に変更します．

* git checkout

　git checkoutは，git branchコマンドによって作成されたブランチ間を移動するコマンドです．ブランチのチェックアウトを行うことにより，作業ディレクトリ内のファイルがそのブランチに保存されているリビジョンに更新され，その後すべての新規コミットはそのブランチに記録されます．このコマンドは，作業を行う開発ラインを選択する手段であると考えることができます．

　前の章において過去のコミットを閲覧する場合のgit checkoutコマンドの使用法を説明しました．ブランチのチェックアウトは，指定されたブランチあるいはバージョンに一致するように作業ディレクトリが更新されるという点では似ていますが，作業ディレクトリに加えられた変更が残っている場合はそれがプロジェクト履歴に保存されるという点が異なります．即ちこのコマンドはリードオンリーの操作ではないのです．

|  |
| --- |
| git checkout <existing-branch> |

　git branchコマンドを使用して作成したブランチのチェックアウトを行うコマンドです．このコマンドを実行すると， <existing-branch> が現在のブランチとなり，それと一致するように作業ディレクトリが更新されます．

|  |
| --- |
| git checkout -b <new-branch> |

新規ブランチ <new-branch> を作成して即時チェックアウトするコマンドです．-bフラグは，git branch <new-branch> コマンドを実行し，続いてgit checkout <new-branch> を実行する便利なフラグです．

|  |
| --- |
| git checkout -b <new-branch> <existing-branch> |

前記のコマンドと同じ機能ですが，ただし現在のブランチではなく <existing-branch> を新規ブランチの基点とします．

* git merge

マージは，Gitにおいて分岐した履歴を戻して統合する手段です．git mergeは，git branchコマンドを使用して作成された独立な複数の開発ラインをひとつのブランチに統合するコマンドです．

ここで，次に説明するすべてのコマンドは現在のブランチへのマージを行うものであることに留意してください．現在のブランチはマージの結果更新されますが，ターゲットブランチ (引数で指定したブランチ) はそのまま残ります．従って，git mergeコマンドは通常現在のブランチを選択するgit checkoutコマンドおよび不要になったターゲットブランチを削除するgit branch -dコマンドと併用されます．

|  |
| --- |
| git merge <branch> |

指定したブランチを現在のブランチにマージするコマンドです．Gitでは，マージアルゴリズムは自動的に選択されます (下で説明します) ．

|  |
| --- |
| git merge --no-ff <branch> |

指定したブランチを現在のブランチにマージしますが，その際に常に (たとえそれが「早送り」可能であっても) マージコミットを作成してマージします．このコマンドは，リポジトリにおいて発生したすべてのマージを記録する場合に有用です．

* git commit –amend

git commit --amendは，直前のコミットを修正する場合に便利なコマンドです．このコマンドを実行すると，全く新たなスナップショットをコミットするのではなく，ステージされた変更内容と直前のコミットとの結合が行なわれます．また，スナップショットに変更を加えずに単に直前のコミットメッセージを編集する場合にも有用です．

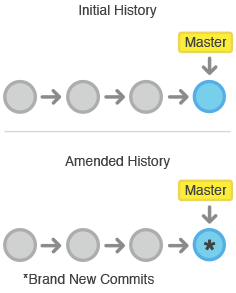


図 6　git commit –amend

ただし直前のコミットの修正とは，それを上書きするのではなく，全く別のコミットで置き換えることを意味します．Gitではそれは上図において星印 (\*) に示すように全く新しいコミットのように見えます．公開リポジトリに対する作業を行う場合はこのことを覚えておく必要があります．

|  |
| --- |
| git commit --amend |

ステージさた変更を直前のコミットと結合し，その結果生成されるスナップショットで直前のコミットを置き換えるコマンドです．ステージエリアに何もない状態でこのコマンドを実行すると，スナップショットを書き換えることなく直前のコミットメッセージの編集を行うことができます．

* git rebase

リベースは，ブランチの基点となるコミットを別のコミットに移動する操作です．一般的な動作を次の図に示します:

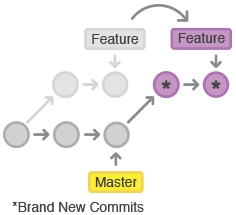


図 7　git rebase

見かけ上は，リベースはあるコミットから他のコミットにブランチを移動する手段に過ぎません．しかし Git の内部では，新たなコミットを生成してそれを移動先のベースコミットに適用することによってこれを行なっており，これは即ち文字通りにプロジェクト履歴の書き換えをしていることになります．ここでは，ブランチそのものは同じものに見えていても，それを構成するコミットは全く異なることを理解することが重要です．

|  |
| --- |
| git rebase <base> |

現在のブランチを <base> にリベースするコマンドで，リベース先としてはすべての種類のコミット参照 (コミットID，ブランチ名，タグ，HEADへの相対参照) を使用することができます．

* git rebase –i

git rebaseを-iフラグを指定して実行するとインタラクティブなリベースセッションが開始されます．インタラクティブなリベースでは，すべてのコミットをそのまま新しいベースに移動するのではなく，対象となる個々のコミットの改変が可能です．これを使用して，既存の一連のコミットの削除，分割，改変を行って履歴を整理することができます．これはちょうどgit commit --amendコマンドの強化版と言えます．

|  |
| --- |
| git rebase -i <base> |

インタラクティブなリベースセッションを使用して現在のブランチを <base> にリベースするコマンドです．このコマンドを実行すると，エディターが開き，リベースする個々のコミットに対するコマンド (下で説明します) の入力が可能となります．ここでのコマンドは，個々のコミットを新しい基点に移動する方法を指定します．また，エディターにおけるコミットの並びを直接編集することによりコミットの順番を並び替えることもできます．

* git reflog

Gitでは，reflogと呼ばれる機能が働いて，ブランチの先端に対する更新の追跡が行われています．これにより，いかなるブランチからもいかなるタグからも参照されていない更新内容であってもその時点に戻ることができます．履歴を書き換えた後であってもreflogにはブランチの過去の状態が記録されており，必要な場合にはそこに戻ることができます．

|  |
| --- |
| git reflog |

ローカルリポジトリのreflogを表示するコマンドです．

|  |
| --- |
| git reflog --relative-date |

相対形式の日付 (例: 2 週間前) でreflogを表示するコマンドです．

* git remote

git remoteは，他のリポジトリとの接続の作成，内容確認，削除を行うコマンドです．リモート接続とは，他のリポジトリへのダイレクトリンクではなく，ブックマークのようなものです．他のリポジトリにリアルタイムアクセスを行うのではなく，非短縮URLへの参照として使用可能な短縮名称として機能します．

例えば，次の図はローカルリポジトリと中央リポジトリとの間およびローカルリポジトリと他の開発者のリポジトリとの間の2つのリモート接続を示したものです．それらをフルURLを用いて参照するのではなく，他のGitコマンドに“origin”および“john”という名称のショートカットを引き渡すことが可能になります．

|  |
| --- |
| git remote |

他のリポジトリへのリモート接続の一覧を表示するコマンドです．

|  |
| --- |
| git remote -v |

上のコマンドと同様ですが，ただし各々の接続のURLが表示されます．

|  |
| --- |
| git remote add <name> <url> |

リモートリポジトリに対する新規接続を作成するコマンドです．作成されると他のGitコマンドにおいて <url> の代わりに <name> を短縮ショートカットとして使用することができます．

|  |
| --- |
| git remote rm <name> |

<name> という名称のリモートリポジトリへの接続を削除するコマンドです．

|  |
| --- |
| git remote rename <old-name> <new-name> |

リモート接続を <old-name> から <new-name> にリネームするコマンドです．

* git fetch

git fetchは，リモートリポジトリからローカルリポジトリにブランチをインポートするコマンドです．インポートされたブランチは，これまで学習してきた通常のローカルブランチとしてではなく，リモートブランチとして保存されます．この機能により，それらをローカルリポジトリに統合する前に変更内容を確認することができます．

|  |
| --- |
| git fetch <remote> |

リポジトリからすべてのブランチをフェッチするコマンドです．このコマンドを実行すると，付随するすべてのコミットおよびファイルもそのリポジトリからダウンロードされます．

|  |
| --- |
| git fetch <remote> <branch> |

上のコマンドと同様の機能を有するコマンドですが，ただしフェッチする対象は指定したブランチのみです．

* git pull

中央リポジトリでの変更のローカルリポジトリへのマージは，Gitベースのコラボレーションワークフローにおいてはよく行われるタスクです．git fetchコマンドとそれに続くgit mergeコマンドを使用してこの操作を行う方法は既に説明しましたが，git pullはこの二つのコマンドをひとつにまとめたコマンドです．

|  |
| --- |
| git pull <remote> |

現在のブランチの指定したリモートにおけるコピーをフェッチして，それを現在のブランチに即時マージします．これは，git fetch <remote> コマンドを実行し，続いてgit merge origin/ <current-branch> .コマンドを実行するのと同等です．

|  |
| --- |
| git pull --rebase <remote> |

上のコマンドと同様ですが，リモートブランチを現在のブランチにマージする際にgit rebaseコマンドを使用します．

* git push

プッシュとは，ブランチをローカルリポジトリからリモートリポジトリに送る操作を意味します．このコマンドはgit fetchと対をなすものですが，フェッチはブランチをローカルリポジトリにインポートする操作であるのに対し，プッシュはブランチをリモートリポジトリにエクスポートする操作です．このコマンドは変更の誤書き込みを起こす可能性があるため，使用の際は注意が必要です．この問題は下で説明します．

|  |
| --- |
| git push <remote> <branch> |

付随するすべてのコミットおよび内部オブジェクトと共に指定したブランチを <remote> にプッシュするコマンドです．このコマンドを実行すると，プッシュ先のリポジトリにローカルブランチが作成されます．変更の誤書き込みを防止するために，Gitではプッシュ先リポジトリにおける統合処理が早送りマージ以外である場合にはプッシュが拒否されます．

|  |
| --- |
| git push <remote> --force |

上のコマンドと同様ですが，早送りマージ以外の場合にも強制的にプッシュが実行されます．プッシュ操作によって何が起こるかを完全に理解している場合以外は--forceフラグを使用してはなりません．

|  |
| --- |
| git push <remote> --all |

すべてのローカルブランチを指定したリモートリポジトリにプッシュするコマンドです．

|  |
| --- |
| git push <remote> --tags |

ブランチをプッシュしただけでは，例え--allフラグが指定されていても，タグは自動的にはプッシュされません．--tagsフラグを指定することにより，すべてのローカルタグをリモートリポジトリに送ることができます[13]．

* 1. **API (Application Programming Interface) の解説**

あるコンピュータプログラム (ソフトウェア) の機能や管理するデータなどを，外部の他のプログラムから呼び出して利用するための手順やデータ形式などを定めた規約のこと．

個々のソフトウェアの開発者が毎回すべての機能をゼロから開発するのは困難で無駄が多いため，多くのソフトウェアが共通して利用する機能は，OSやミドルウェアなどの形でまとめて提供されている．そのような汎用的な機能を呼び出して利用するための手続きを定めたものがAPIで，個々の開発者はAPIに従って機能を呼び出す短いプログラムを記述するだけで，自分でプログラミングすることなくその機能を利用したソフトウェアを作成することができる．

近年ではネットワークを通じて外部から呼び出すことができるAPIを定めたソフトウェアも増えており，遠隔地にあるコンピュータの提供する機能やデータを取り込んで利用するソフトウェアを開発することができる[15]．

* 1. **GitHub APIのコード**

　GitHub APIとは，GitHub上に存在するデータを取得する際に必要なコードである．GitHub APIのコードは機能ごとに設定されている．以下に，本研究で使用したGitHub APIを記載する．

* Issues

　ユーザやリポジトリ，IssuesのIDなどを指定することで，指定のIssuesを表示する．

|  |
| --- |
| GET /issues |

ユーザのIssues全てを一覧表示する．

|  |
| --- |
| GET /user/issues |

指定されたユーザのIssues全てを一覧表示する．

|  |
| --- |
| GET /repos/:owner/:repo/issues |

指定されたユーザのリポジトリのIssues全てを一覧表示する．

|  |
| --- |
| GET /repos/:owner/:repo/issues/:number |

指定されたユーザのリポジトリのIssuesをIDから選択し，表示する．

* Commit

　ユーザやリポジトリ，CommitのSHAなどを指定することで，指定のCommit履歴を表示する．SHA (Secure Hash Algorithm) とは，一群の関連したハッシュ関数である．

|  |
| --- |
| GET /repos/:owner/:repo/commits |

　指定されたユーザのリポジトリのCommit履歴を一覧表示する．

|  |
| --- |
| GET /repos/:owner/:repo/commits/:sha |

　指定されたユーザのリポジトリのCommit履歴をSHAから選択し，表示する[16]．

第4章

開発・調査

# **開発・調査**

* 1. **本章の構成**

　本章では，OSS開発プロジェクトの成功の成否に関連するパターンを発見するために調査した対象，及び調査方法を記述する．また，GitHub上からIssuesを取得するプログラムのコードと解説を記述する．

* 1. **調査対象**

　本研究において調査対象となるデータの記述，及びデータを取得する対象のプロジェクトのユーザ名，リポジトリ名を記載する．

* + 1. **調査対象データ**

本研究では，OSS開発プロジェクトの成功の成否に関連するパターンを発見するためにプロジェクトの分類を行う．プロジェクトを分類するために，以下のデータをGitHub上から取得する．

表 3　調査対象データ

|  |  |
| --- | --- |
| データ名 | 解説 |
| Issues  (イシュー) | 開発中に発生したバグや課題を記述し，登録する機能．1つのバグや課題に1つのIssuesが割り当てられる．プロジェクトメンバ間での共有が可能であり，コメント，削除などの操作ができる．課題が未完了のIssuesをOpenIssuesと呼び，完了済みのIssuesをClosedIssuesと呼ぶ．チケット駆動開発のチケットと同様な役割を持つ．  プロジェクト毎に総Issues数とOpenIssues数，ClosedIssues数を取得する．Star数とどのような関係があるのかを調査する． |
| Commits  (コミット) | ステージングエリアに追加した変更をGitディレクトリ上に変更履歴として反映する．  プロジェクト毎にCommits履歴を取得する．プロジェクトの開始時期を最初にCommitsがされた日付と定める． |
| Star  (スター) | ソーシャルブックマークと同様な機能であり，スターを付けたリポジトリを即座に閲覧できるようになる．  本研究では他者からStarされた数をStar数と呼称する．プロジェクト毎にStar数を取得し，調査対象プロジェクトを選定する際の判断基準とする．また，Star数がプロジェクトの成功要因であるかを調査する． |
| Contributors  (コントリビュータ) | GitHub上のプロジェクトへの寄与貢献者．Pull Requestの承認やリポジトリの各種操作を行える権限を持つ．  プロジェクト毎にContributors数を取得する．Star数とどのような関係があるのかを調査する． |
| Fork  (フォーク) | GitHub上で公開されているリポジトリを自分のリポジトリとして複製する．  プロジェクト毎にForkされた数を取得する．Star数とどのような関係があるのかを調査する． |
| Releases  (リリース) | GitHub上で開発したソフトウェアをGitHub上で配布できる機能である．  プロジェクト毎にリリースバージョンを取得する．リリースされているバージョンにより，プロジェクト成功の成否を判断する． |
| Issuesを完了するまでの所要時間の平均 | Issuesを完了するまでの所要時間の平均である．調査対象のプロジェクトから算出する． |
| Issuesを完了するまでの所要時間の標準偏差 | Issuesを完了するまでの所要時間の標準偏差である．調査対象のプロジェクトから算出する． |
| 線形近似曲線の式 | 時系列データにフィットする線形式の係数である． |

* + 1. **調査対象プロジェクト**

　本研究では，OSS開発プロジェクトの成功の成否に関連するパターンを発見するために，以下の100件のプロジェクトを解析する．100件のプロジェクトは2種類に大別し，Star数ランキングの上位50件のプロジェクト群とランダムに選択した50件のプロジェクト群とする．これらのプロジェクトは，2013-12-19時点での，Star数ランキングの上位5000件に入ったプロジェクトをまとめたデータから選択した[17]．以下に調査対象とした100件のプロジェクトのユーザ名とリポジトリ名を記載する．

表 4　調査対象プロジェクト　上位50件

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ユーザ名 | リポジトリ名 |  | ユーザ名 | リポジトリ名 |
| twbs | bootstrap |  | TryGhost | Ghost |
| joyent | node |  | ivaynberg | select2 |
| angular | angular.js |  | nnnick | Chart.js |
| mbostock | d3 |  | ariya | phantomjs |
| vhf | free-programming-books |  | emberjs | ember.js |
| FortAwesome | Font-Awesome |  | kennethreitz | requests |
| h5bp | html5-boilerplate |  | plataformatec | devise |
| rails | rails |  | mitsuhiko | flask |
| bartaz | impress.js |  | less | less.js |
| Homebrew | homebrew |  | airbnb | javascript |
| robbyrussell | oh-my-zsh |  | caolan | async |
| adobe | brackets |  | facebook | hhvm |
| jashkenas | backbone |  | defunkt | jquery-pjax |
| moment | moment |  | jashkenas | coffeescript |
| zurb | foundation |  | mathiasbynens | dotfiles |
| blueimp | jQuery-File-Upload |  | mozilla | pdf.js |
| hakimel | reveal.js |  | diaspora | diaspora |
| daneden | animate.css |  | jquery | jquery-mobile |
| jekyll | jekyll |  | ajaxorg | ace |
| harvesthq | chosen |  | Leaflet | Leaflet |
| AFNetworking | AFNetworking |  | h5bp | Effeckt.css |
| necolas | normalize.css |  | gruntjs | grunt |
| gitlabhq | gitlabhq |  | addyosmani | backbone-fundamentals |
| resume | resume.github.com |  | usablica | intro.js |
| Modernizr | Modernizr |  | xdissent | ievms |

　2013-12-19時点での，Star数ランキングの上位5000件に入ったプロジェクトをまとめたデータから，Star数上位50件のプロジェクトを選択した．選択した50件のプロジェクトの内，リポジトリが削除されているプロジェクトが確認された場合は，そのプロジェクトを無視し，新たに一つ下位のプロジェクトを選択する．

表 5　調査対象プロジェクト　ランダム50件

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ユーザ名 | リポジトリ名 |  | ユーザ名 | リポジトリ名 |
| angular | angular-seed |  | kennytm | iphone-private-frameworks |
| rstacruz | jquery.transit |  | thefaj | OpenFlow |
| loopj | android-async-http |  | nryoung | algorithms |
| keithw | mosh |  | ftlabs | ftscroller |
| bup | bup |  | jonase | kibit |
| rigoneri | syte |  | zendtech | ZendOptimizerPlus |
| goldfire | howler.js |  | k4rthik | git-cal |
| LeaVerou | prefixfree |  | scrapy | scrapely |
| fatfreecrm | fat\_free\_crm |  | manuels | texlive.js |
| presidentbeef | brakeman |  | jnicklas | turnip |
| John-Lluch | SWRevealViewController |  | sellout | emacs-color-theme-solarized |
| aphyr | riemann |  | symfony-cmf | symfony-cmf |
| tenXer | xcharts |  | ttscoff | Slogger |
| MugunthKumar | MKStoreKit |  | jbrewer | Responsive-Measure |
| suprb | Nested |  | remy | polyfills |
| 1602 | jugglingdb |  | mapstraction | mxn |
| NYTimes | backbone.stickit |  | technoweenie | coffee-resque |
| metamx | druid |  | kylemcdonald | ofxFaceTracker |
| radar | guides |  | marcj | jquery-selectBox |
| opauth | opauth |  | rezoner | CanvasQuery |
| GravityLabs | goose |  | snwau | SWSnapshotStackView |
| RobinHerbots | jquery.inputmask |  | yangmeyer | CPAnimationSequence |
| tschellenbach | Stream-Framework |  | thoughtbot | hoptoad\_notifier |
| rochal | jQuery-slimScroll |  | azer | onejs |
| rack | rack-contrib |  | dojo | dojo-oldmirror |

　2013-12-19時点での，Star数ランキングの上位5000件に入ったプロジェクトをまとめたデータから，ランダムに50件のプロジェクトを選択した．選択にはExcelのRAND関数=RAND()を使用した．選択した50件のプロジェクトの内，リポジトリが削除されているプロジェクトが確認された場合は，そのプロジェクトを無視し，再度ランダムに選択を行う．

* 1. **調査環境の構築**

　調査に必要である調査環境の構築過程を記述する．調査環境のOSはUbuntu (ウブントゥ) である．

* + 1. **Ubuntuの解説**

　Ubuntu (ウブントゥ) とは，コミュニティ により開発されているオペレーティングシステムです．ラップトップ，デスクトップ，そしてサーバーに利用することができます．Ubuntuには，家庭・学校・職場で必要とされるワープロやメールソフトから，サーバーソフトウェアやプログラミングツールまで，あらゆるソフトウェアが含まれています．

　Ubuntuは現在，そして将来に渡って無償で提供されます．ライセンス料を支払う必要はありません．Ubuntuをダウンロードすれば，友達や家族と，あるいは学校やビジネスに，完全に無料で利用できます．

　私たちは，新しいデスクトップおよびサーバーを6ヶ月ごとにリリースすることを宣言しています．これにより，オープンソースの世界が提供する最新の優れたアプリケーションを常に利用できるようにしています． Quantal Quetzal 12.10 2012年10月18日 2014年4月

　Ubuntuは，セキュリティに配慮して設計されています．デスクトップおよびサーバーの無償セキュリティアップデートが，少なくとも9ヶ月間に渡って提供されます．長期サポート(LTS)版を利用すれば，5年間に渡りセキュリティアップデート提供されます．もちろん，LTS版を利用するために追加の費用は必要ありません．すべての人が無償という同じ条件で，私たちの精一杯の成果を利用することができます．Ubuntuを新しいバージョンにアップグレードする場合も，常に無償です．

　Ubuntuのインストールイメージにはデスクトップ環境がひと通り含まれています．さらに，オンラインでソフトウェアを追加することができます．

　グラフィカルインストーラにより，素早く簡単にインストールして使い始めることができます．標準的なインストールにかかる時間は10〜20分未満です．十分に高速な環境であれば，インストールが5分程度で終了することもあります．

　一度システムをインストールすれば，インターネット，ドローイング，グラフィックス，そしてゲームといったアプリケーションがすぐに使えるようになります．

　Ubuntuサーバーでは，ユーザーがセットアップしたものだけが動作し，それ以外はインストールされません[18]．

* + 1. **調査環境の構築と処理テスト**

1. curlのインストール

|  |
| --- |
| sudo apt-get install curl |

　Ubuntuにcurlをインストールする．

1.1)　Issuesの取得テスト

|  |
| --- |
| curl –s –u ユーザ名:パスワード “https://api.github.com/repos/less/less.js/issues” |

　GitHub APIを使用し，プロジェクトless/less.jsからIssuesを取得する．使用するGitHub APIは GET /repos/:owner/:repo/issues である．

1. GitHubログイン情報入力の省略

|  |
| --- |
| echo ‘ユーザ名:パスワード’ > github.passwd |

　ログイン情報を入力する作業を省略するため，ユーザ名とパスワードをファイルgithub.passwdに記述する．

|  |
| --- |
| Chmod 600 github.passwd |

　ファイルgithub.passwdに読み込み権限と書き込み権限を付加する．所有者以外はファイルgithub.passwdに対する操作が不可能である．

2.1)　github.passwdを使用してのIssuesの取得テスト

|  |
| --- |
| cat github.passwd |

　ユーザ名とパスワードがファイルgithub.passwdに正しく記述されているのかを確認する．

|  |
| --- |
| Curl –s –u $(cat github.passwd) “https://api.github.com/repos/less/less.js/issues” |

　GitHub APIとgithub.passwdを使用し，プロジェクトless/less.jsからIssuesを取得する．使用するGitHub APIはGET /repos/:owner/:repo/issues である．

|  |
| --- |
| Curl –s –u $(cat github.passwd) “https://api.github.com/repos/less/less.js/issues” | grep title |

　｜ (パイプ) コマンドとgrepコマンドを使用し，API呼び出し結果の内容を確認する．| grep title により，Issuesのタイトルを確認できる．

1. Pythonとrequestsのインストール

|  |
| --- |
| sudo apt-get install python python-setuptools |

　UbuntuにPythonをインストールする．

|  |
| --- |
| Sudo easy\_install requests |

　UbuntuにHTTPアクセスのためのrequestsをインストールする．

1. api.pyの開発

|  |
| --- |
| #!/usr/bin/python  # coding: UTF-8  import sys, json, requests  #GitHubのログイン情報をファイルから取得する  #TODO:パスワードに「:」を使っているとダメ  tmp = open(‘github.passwd’).readline().rstrip(‘\n’).split(‘:’);  username = tmp[0]  password = tmp[1]  #print >> sys.stderr, username,password  #APIのURLはコマンドライン引数で与える  url = sys.argv[1]  count = 0  while (url is not None):  print >> sys.stderr, url  r = requests.get(url, auth=(username, password))  print >> sys.stderr, r.headers[‘status’],  items = r.json()[‘items’] if ‘items’ in r.json() else r.json()  for item in items:  count = count + 1  print json.dumps(item)  if (r.links.has\_key(‘next’)):  url = r.links[‘next’][‘url’]  else:  url = None  print >> sys.stderr, count, ‘items’ |

　Python形式で開発したプログラムである．GitHubログイン情報入力の省略，および各種設定を行う．

4.1)　api.pyを使用してのOpenIssues取得テスト

|  |
| --- |
| python api.py “https://api.github.com/repos/less/less.js/issues?per\_page=100”  > openissues.txt |

　GitHub APIを使用し，プロジェクトless/less.jsのOpenなIssuesを全て取得する．結果は“openissues.txt”に保存する．使用するGitHub APIはGET /repos/:owner/:repo/issues である． ?per\_page=100 により，一度に取得するデータを最大にしておくことで，API使用可能回数の消費を抑える．

4.2)　api.pyを使用してのClosedIssues取得テスト

|  |
| --- |
| python api.py “https://api.github.com/repos/less/less.js/issues?per\_page=100&state=closed” > closedissues.txt |

　GitHub APIを使用し，プロジェクトless/less.jsのClosedなIssuesを全て取得する．結果は“closedissues.txt”に保存する．使用するGitHub APIはGET /repos/:owner/:repo/issues である．

4.3)　api.pyを使用してのCommits履歴取得テスト

|  |
| --- |
| python api.py “https://api.github.com/repos/less/less.js/commits?per\_page=100”  > commits.txt |

　GitHub APIを使用し，プロジェクトless/less.jsのCommits履歴をすべて取得する．結果は“commits.txt”に保存する．使用するGitHub APIはGET /repos/:owner/:repo/ commits である．

1. jqのインストール

|  |
| --- |
| sudo apt-get install jq |

　UbuntuにJSON形式のデータを扱うためにjqをインストールする．

|  |
| --- |
| Chmod +x jq |

　jqを実行するために，jqに実行権限を付加する．

5.1)　Issuesのタイトルと作成日時の一覧表示テスト

|  |
| --- |
| jq '.id,.title,.created\_at' openissues.txt |

　OpenIssues取得テストで作成した“openissues.txt”に保存されているIssuesのタイトルと作成日時を一覧表示する．Issuesのタイトルは，重複を避けるためにIDを付けて表示する．

5.2)　Commits IDとCommits日時の一覧表示テスト

|  |
| --- |
| jq '.commit.committer.date,.sha' commits.txt |

　Commits履歴取得テストで作成した“commits.txt”に保存されているCommits IDとCommits日時を一覧表示する．

* 1. **調査方法**

本研究では，OSS開発プロジェクトの成功の成否に関連するパターンを発見することを目的とする．そのために，プロジェクトの分類と重回帰分析，プロジェクト成功の成否の調査を行う．

* + 1. **プロジェクト成功の成否の調査**

本研究では，Star数がプロジェクトの成功要因であると仮定する．この仮定の真偽を検証するため，Star数ランキング上位50件のプロジェクトとランダムに選択した50件のプロジェクトからStar数とリリースされたソフトウェアのバージョンを取得し，相関関係を調査する．成功の成否は，リリースされたソフトウェアのバージョンから判断する．Ver. 1.0以上をリリースしているプロジェクトを成功とし，Ver. 1.0未満でリリースが止まっているプロジェクトを失敗とする．GitHub上でリリースしていないプロジェクトは，GitHub外でリリースしている可能性があるため，検索エンジンGoogleを使用して調査する．ソフトウェアをダウンロードできるサイトなどが見つかれば成功とし，見つからなければ失敗とする．以下にStar数とリリースバージョン，成功の成否を調査する方法を記述する．

1. Star数の調査

　Star数の調査は手動で行う．

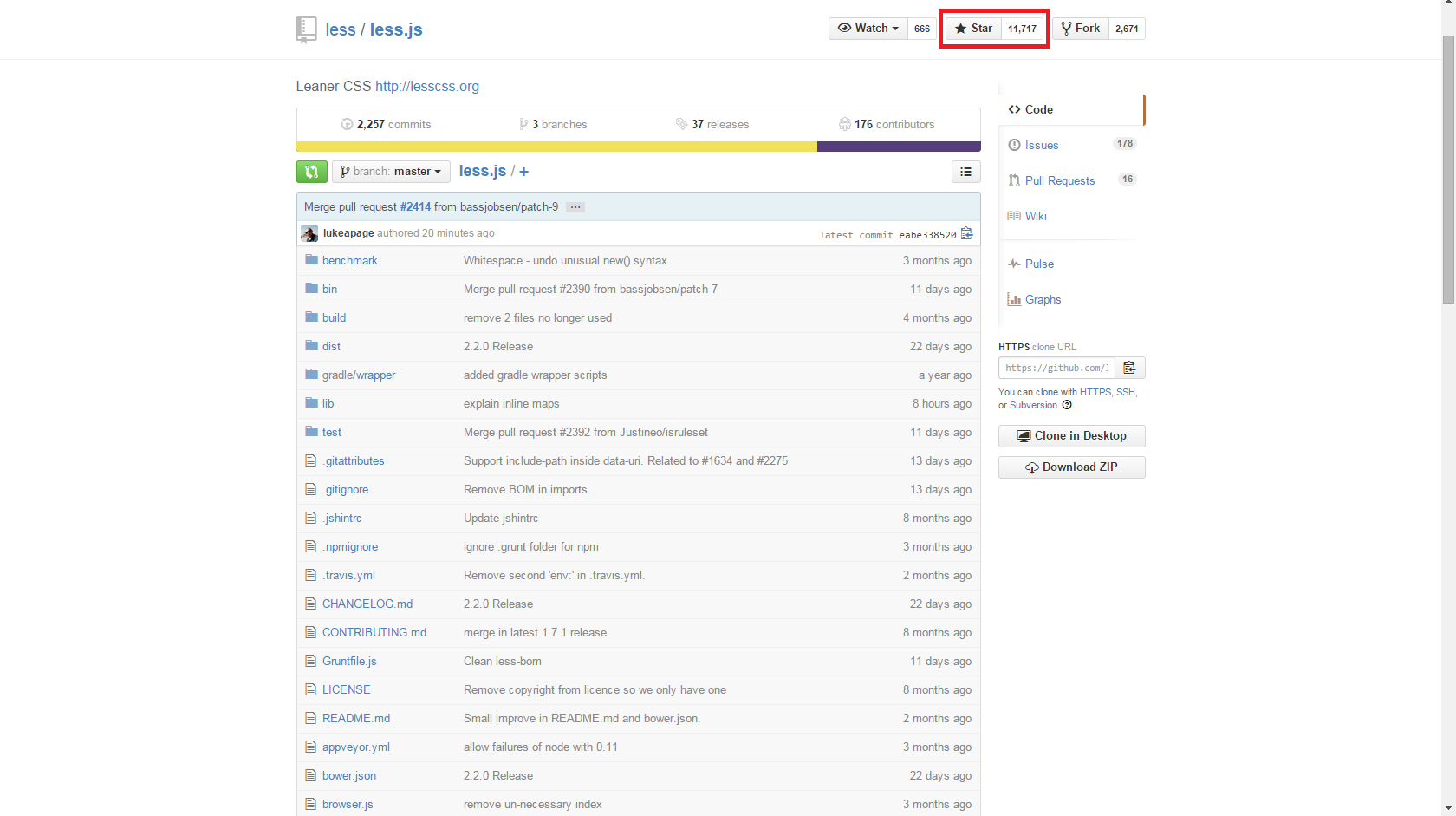


図 8　Star数の調査

　図8はプロジェクトless/less.jsのStar数の調査結果画面である．赤枠に囲まれている数値がStar数である．Star数ランキング上位50件のプロジェクトから取得したStar数を“Rank.csv”に保存する．ランダムに選択した50件のプロジェクトから取得したStar数を“Random.csv”に保存する．

1. リリースバージョンの調査

　リリースバージョンの調査は手動で行う．

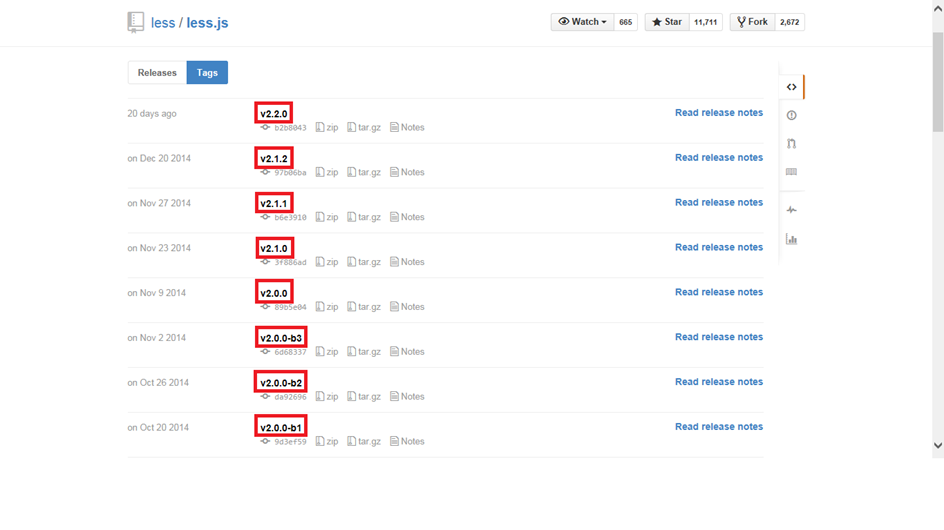


図 9　リリースバージョンの調査

図9はプロジェクトless/less.jsのリリースバージョンの調査結果画面である．赤枠に囲まれている数値がリリースバージョンであり，プロジェクトless/less.jsの最新バージョンは2.2.0である．Star数ランキング上位50件のプロジェクトから取得したリリースバージョンを“Rank.csv”に保存する．ランダムに選択した50件のプロジェクトから取得したリリースバージョンを“Random.csv”に保存する．

1. 成功の成否の調査

“rank.csv” と“random.csv”から，それぞれのプロジェクト群の成否表を作成する．

表 6　Star数ランキング上位50件のプロジェクト (Rank.csv)

|  |  |  |
| --- | --- | --- |
| Star数＼リリースバージョン | ver. 1.0 未満 (失敗) | ver. 1.0 以上 (成功) |
| 0 ~ 10000 | 4 | 6 |
| 10001 ~ 20000 | 4 | 24 |
| 20001 ~ 30000 | 1 | 6 |
| 30001 ~ | 1 | 4 |
| 合計プロジェクト件数 | 10 | 40 |

　表6はStar数ランキング上位50件のプロジェクトの成否表である．成功しているプロジェクトは50件中40件であり，失敗しているプロジェクトは50件中10件である．

表 7　ランダムに選択した50件のプロジェクト (Random.csv)

|  |  |  |
| --- | --- | --- |
| Star数＼リリースバージョン | ver. 1.0 未満 (失敗) | ver. 1.0 以上 (成功) |
| 0 ~ 500 | 3 | 3 |
| 501~1000 | 9 | 13 |
| 1001~1500 | 3 | 4 |
| 1501~ | 5 | 10 |
| 合計プロジェクト件数 | 20 | 30 |

　表7はランダムに選択した50件のプロジェクトの成否表である．成功しているプロジェクトは50件中30件であり，失敗しているプロジェクトは50件中20件である．

　Star数が増加すると，成功しているプロジェクトが増加し，失敗しているプロジェクトが減少する．Star数が減少すると，成功しているプロジェクトが減少し，失敗しているプロジェクトが増加する．この結果から，Star数がプロジェクトの成功要因であることを証明できる．

* + 1. **プロジェクト開始時期の調査**

GitHub上のプロジェクトは，GitHubに登録される前からプロジェクトが開始されている可能性がある．そのため，プロジェクトの開始時期を定める必要がある．本研究では，プロジェクトの開始時期を最初にCommitsがされた日付と定める．以下にGitHub上のデータから，Commits履歴を調査する方法を記述する．

|  |
| --- |
| python api.py "https://api.github.com/repos/ユーザ名/リポジトリ名/commits?per\_page=100"  > ユーザ名-リポジトリ名-commits.txt |

　api.pyとGitHub APIを使用し，調査対象であるプロジェクトのコミット履歴を全て取得する．結果は“ユーザ名-リポジトリ名-commits.txt”に保存する．ファイルの命名規則を分かりやすくするため，ファイル名は“ユーザ名-リポジトリ名-commits”で統一する．保存した結果から，最初にCommitsがされた日付を特定し，その日付をプロジェクト開始時期と定める．日付はdateで検索して特定する．



図 10　Commits履歴の調査結果

　図10はプロジェクトless/less.jsのCommits履歴を保存した“less-less.js-commits.txt”を開いた時の画面である．プロジェクトless/less.jsの最初にCommitsがされた日付は"date": "2010-02-23T18:39:05Z"である．よってプロジェクトless/less.jsのプロジェクト開始時期を2010-02-23と定める．

* + 1. **Issuesの調査**

　プロジェクト分類のため，100件のプロジェクトからOpenIssuesとClosedIssuesを調査する．調査する項目は，OpenIssuesとClosedIssuesのそれぞれの発行日時と累積数，及びClosedIssuesの完了日時である．以下にGitHub上のデータから，Issues数の時間変化を調査する方法を記述する．

1. OpenIssuesとClosedIssuesの取得

|  |
| --- |
| python api.py "https://api.github.com/repos/ユーザ名/リポジトリ名/issues?per\_page=100"  > ユーザ名-リポジトリ名-openissues.txt |

|  |
| --- |
| python api.py "https://api.github.com/repos/ユーザ名/リポジトリ名/issues?per\_page=100&state=closed" > ユーザ名-リポジトリ名-closedissues.txt |

　api.py とGitHub APIを使用し，調査対象であるプロジェクトのOpenIssuesとClosedIssuesを取得する．結果はそれぞれ“ユーザ名-リポジトリ名-open (closed) issues.txt”に保存する．ファイルの命名規則を分かりやすくするため，ファイル名は“ユーザ名-リポジトリ名-open (closed) issues”で統一する．使用するGitHub APIはGET /repos/:owner/:repo/issues である．

1. Open・Closedの情報を取得

|  |
| --- |
| ./jq '.created\_at' ユーザ名-リポジトリ名-openissues.txt | awk '{ printf("%s open\n", $0); }' > ユーザ名-リポジトリ名-issues.tmp |

　“ユーザ名-リポジトリ名-openissues.txt”からOpenなIssuesの発行日時と件数を取得し，“ユーザ名-リポジトリ名-issues.tmp”に保存する．

|  |
| --- |
| ./jq '.created\_at,.closed\_at' ユーザ名-リポジトリ名-closedissues.txt | awk '{ if (NR % 2 == 1) printf("%s open\n", $0); else printf("%s close\n", $0); }' >> ユーザ名-リポジトリ名-issues.tmp |

　“ユーザ名-リポジトリ名-closedissues.txt”からClosedなIssuesの発行日時と完了日時，件数を取得し，“ユーザ名-リポジトリ名-issues.tmp”に追記する．

1. Issuesの並び替え

|  |
| --- |
| sort ユーザ名-リポジトリ名-issues.tmp > ユーザ名-リポジトリ名-issues.txt |

　“ユーザ名-リポジトリ名-issues.tmp”に保存されているIssuesを時間で並び替え，“ユーザ名-リポジトリ名-issues.txt”に保存する．

1. Issuesの累積数の算出

|  |
| --- |
| awk 'BEGIN { openissues=0; closedissues=0; } $2=="open" { openissues++; } $2=="close" { closedissues++; } { printf("%s,%d,%d\n", $1, openissues, closedissues) }' ユーザ名-リポジトリ名-issues.txt > ユーザ名-リポジトリ名-issues.csv |

　“ユーザ名-リポジトリ名-issues.txt”からOpenIssuesとClosedIssuesのそれぞれの累積数を算出し，“ユーザ名-リポジトリ名-issues.csv”に保存する．

　OpenIssues数とClosedIssues数は，プロジェクトの分類と重回帰分析の際に必要となる．そのため，Star数ランキング上位50件のプロジェクトから取得したOpenIssues数とClosedIssues数を“RankData.csv”に保存する．ランダムに選択した50件のプロジェクトから取得したOpenIssues数とClosedIssues数を“RandomData.csv”に保存する．

1. Issuesの線形式の取得

　プロジェクト分類の際に必要となる時系列データにフィットする線形式の係数を取得する． 線形式の係数は，Issuesの累積数をグラフ化し，線形近似曲線の式を表示することで取得する．取得先は本項の4) で作成したファイル“ユーザ名-リポジトリ名-issues.csv”である．

表 8　Issuesの累積数と経過日数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Issuesの  発行・完了日時 | 日数 | 経過日数 | OpenIssues累積数 | ClosedIssues累積数 |
| 2010/2/23 | 1 | 1 | 0 | 0 |
| 2010/3/22 | 27 | 28 | 1 | 0 |
| 2010/3/23 | 1 | 29 | 2 | 0 |
| 2010/4/4 | 12 | 41 | 3 | 0 |
| 2010/4/7 | 3 | 44 | 4 | 0 |
| ~ | ~ | ~ | ~ | ~ |
| 2014/6/24 | 0 | 1583 | 2069 | 1822 |
| 2014/6/24 | 0 | 1583 | 2070 | 1822 |
| 2014/6/25 | 1 | 1584 | 2071 | 1822 |
| 2014/6/25 | 0 | 1584 | 2071 | 1823 |
| 2014/6/25 | 0 | 1584 | 2072 | 1823 |

　表8はプロジェクトless/less.jsのIssuesの累積数と経過日数をまとめたファイル“less-less.js-issues.csv”から一部抜粋したデータである．経過日数はIssuesの発行・完了日時から算出する．算出する際，本節第1項で定めたプロジェクト開始時期を計算に加える．このプロジェクトのプロジェクト開始時期は2010-02-23である．

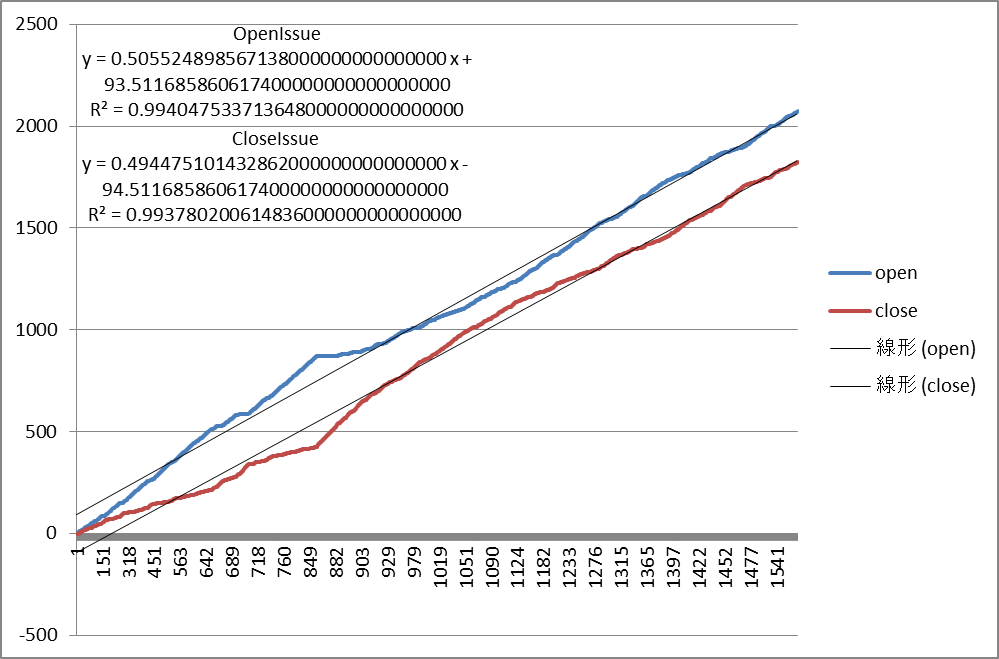


図 11　線形近似曲線の式

　図11はプロジェクトless/less.jsのIssuesの累積数のグラフである．データ選択は，縦軸にOpenIssues累積数とClosedIssues累積数，横軸に経過日数を選択する．近似曲線は，線形近似曲線を選択する．OpenIssuesの線形式はy = 0.505524898567138 x + 93.5116858606174である．ClosedIssuesの線形式はy = 0.494475101432862 x - 94.5116858606174である．

　線形式の係数は，プロジェクトの分類と重回帰分析の際に必要となる．そのため，Star数ランキング上位50件のプロジェクトから取得した線形式の係数を“RankData.csv”に保存する．ランダムに選択した50件のプロジェクトから取得した線形式の係数を“RandomData.csv”に保存する．

1. “IssuesがOpenされてからClosedされるまでの時間”の算出

|  |
| --- |
| jq '.id,.created\_at,.closed\_at,.title' ユーザ名-リポジトリ名-closedissues.txt | awk '{printf("%s", $0); if (NR % 4 == 0) printf("\n"); else printf(","); }' > ユーザ名-リポジトリ名-issuestime.csv |

　本項の1) で作成した“ユーザ名-リポジトリ名-closedissues.txt”からIssuesの発行日時と完了日時，IDとタイトルの一覧を作成し，“ユーザ名-リポジトリ名- issuestime.csv”に保存する．“IssuesがOpenされてからClosedされるまでの時間”の算出式は，= (完了日時–発行日時)\*24 である．

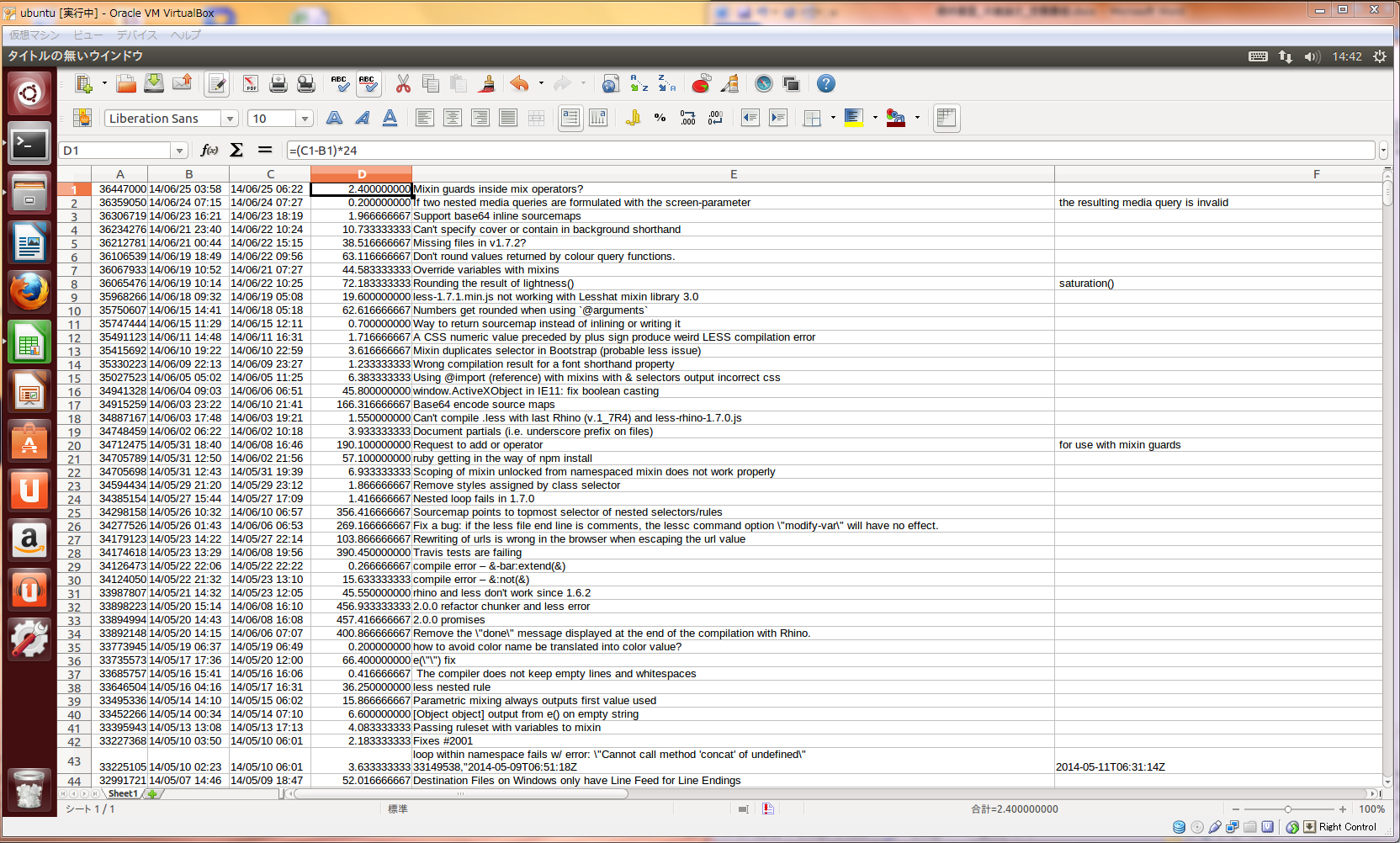


図 12　Issuesの調査結果

図12はプロジェクトless/less.jsのIssuesの調査結果を保存した“less-less.js-issuestime.csv”をCalcで開いた時の画面である．D1に= (C1-B1)\*24 と入力し，D列全体にコピーして“IssuesがOpenされてからClosedされるまでの時間”を算出する．

1. Issuesを完了するまでの所要時間の平均と標準偏差の算出

　本項の6) で算出した“IssuesがOpenされてからClosedされるまでの時間”から算出する．平均は=AVERAGE() で算出し，標準偏差は=STDEV() で算出する．

　Issuesを完了するまでの所要時間の平均と標準偏差は，プロジェクトの分類と重回帰分析の際に必要となる．そのため，Star数ランキング上位50件のプロジェクトから取得したIssuesを完了するまでの所要時間の平均と標準偏差を“RankData.csv”に保存する．ランダムに選択した50件のプロジェクトから取得したIssuesを完了するまでの所要時間の平均と標準偏差を“RandomData.csv”に保存する．

* + 1. **Star数，Contributors数，Fork数の調査**

重回帰分析の際に必要となるStar数，Contributors数，Fork数を調査する．調査は手動で行う．

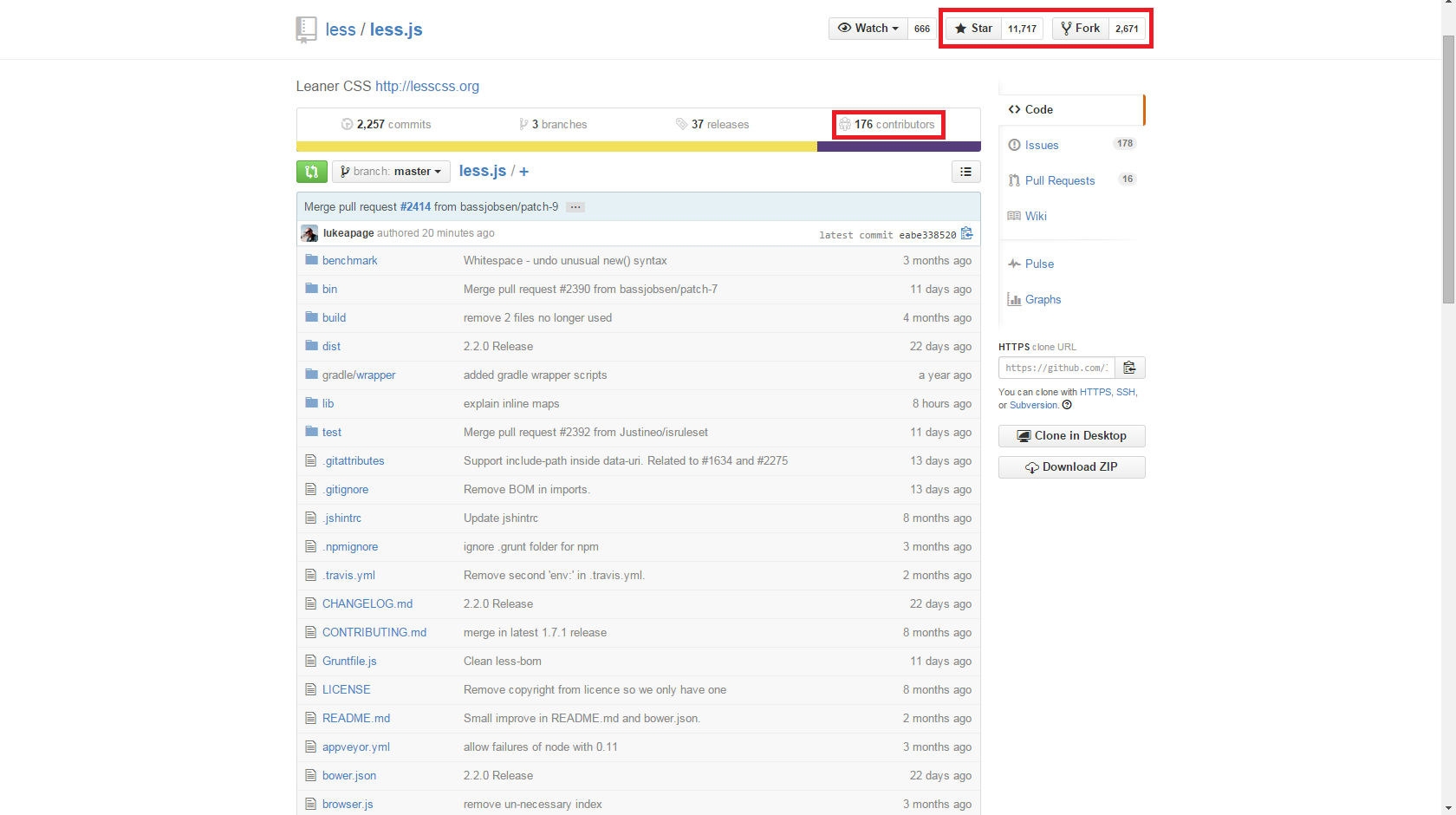


図 13　 Star数，Contributors数，Fork数の調査結果

　図13はプロジェクトless/less.jsのStar数，Contributors数，Fork数の調査結果画面である．赤枠に囲まれている数値を取得する． Star数ランキング上位50件のプロジェクトから取得したStar数，Contributors数，Fork数を“RankData.csv”に保存する．ランダムに選択した50件のプロジェクトから取得したStar数，Contributors数，Fork数を“RandomData.csv”に保存する．

* + 1. **プロジェクトの分類**

100件のプロジェクトを幾つかのパターンに分類し，成功の成否に関連するパターンを調査する．分類には，統計解析ソフト“R”を使用する．階層クラスター分析と非階層クラスター分析，自己組織化マップの作成をし，得られた結果から分類をする．以下に“RankData.csv”に保存しているデータを使用してプロジェクトを分類する方法を記述する．

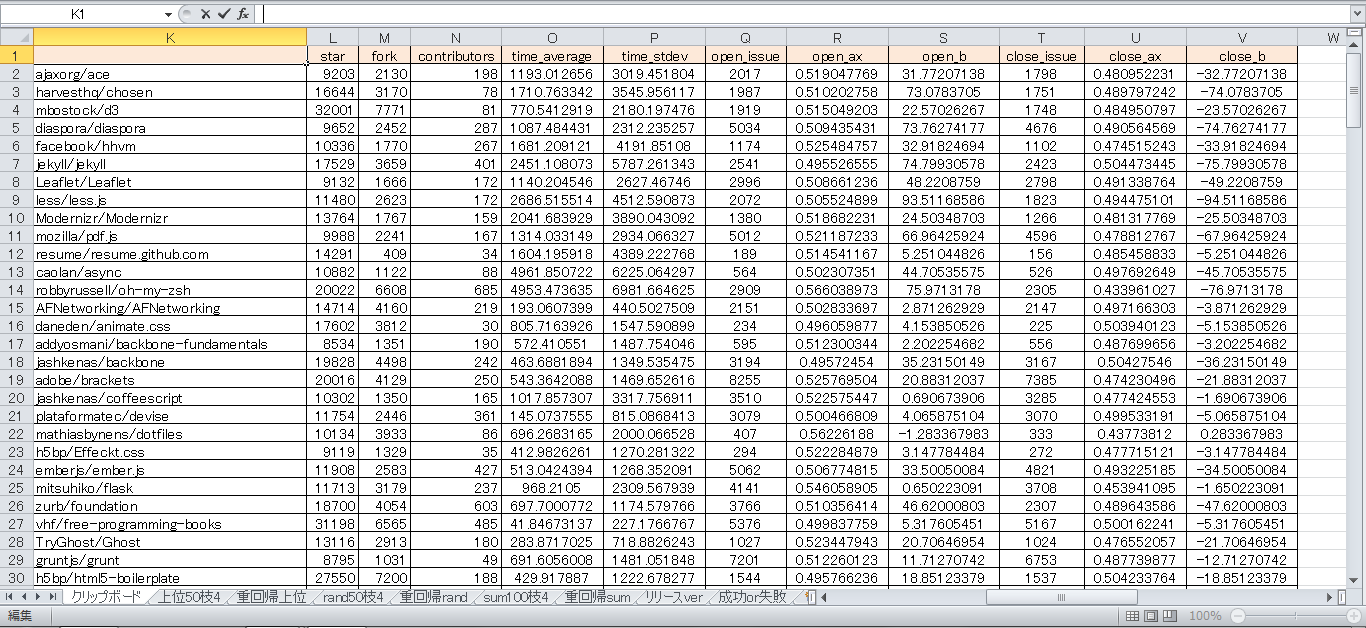


図 14　RankData.csv

　図14は“RankData.csv”の表示画面である．Star数ランキング上位50件のプロジェクトのデータをこのファイルに保存している．保存しているデータは下記の表9に記述する．

表 9　分析に使用する変数

|  |  |
| --- | --- |
| 変数名 | 解説 |
| star | プロジェクトから取得したStar数である． |
| fork | プロジェクトから取得したFork数である． |
| contributors | プロジェクトから取得したContributors数である． |
| time\_average | プロジェクトから取得したIssuesを完了するまでの所要時間の平均である． |
| time\_stdev | プロジェクトから取得したIssuesを完了するまでの所要時間の標準偏差である． |
| open\_issue | プロジェクトから取得したOpenIssues数である． |
| open\_ax | プロジェクトから取得したOpenIssuesの線形式の傾きである． |
| open\_b | プロジェクトから取得したOpenIssuesの線形式の切片である． |
| close\_issue | プロジェクトから取得したClosedIssues数である． |
| close\_ax | プロジェクトから取得したClosedIssuesの線形式の傾きである． |
| close\_b | プロジェクトから取得したClosedIssuesの線形式の切片である． |

　表9は分析に使用する変数をまとめたものである．分かりやすいように変数の名称を表9のとおりにする．

1. データの読み込み

|  |
| --- |
| Data1 <- read.csv("RankData.csv",header=T,row.names=1) |

　分類方法を解説するため，Star数ランキング上位50件のプロジェクトのデータを読み込む．“RankData.csv”のデータを読み込み，データセット“Data1”に保存する．

1. 分類に不要な変数の削除

|  |
| --- |
| Data1$contributors <- NULL |
| Data1$fork <- NULL |
| Data1$star <- NULL |

　データセット“Data1”から分類に不要な変数であるStar数，Contributors数，Fork数を削除する．

1. 変数の標準化

|  |
| --- |
| .Z <- scale(Data1[,c("close\_ax","close\_b","close\_issue","open\_ax","open\_b","open\_issue","time\_average","time\_stdev")]) |

　データセット“Data1”の変数を標準化する．標準化した変数をデータセット“.Z”に保存する．

|  |
| --- |
| Data1$Z.close\_ax <- .Z[,1] |
| Data1$Z.close\_b <- .Z[,2] |
| Data1$Z.close\_issue <- .Z[,3] |
| Data1$Z.open\_ax <- .Z[,4] |
| Data1$Z.open\_b <- .Z[,5] |
| Data1$Z.open\_issue <- .Z[,6] |
| Data1$Z.time\_average <- .Z[,7] |
| Data1$Z.time\_stdev <- .Z[,8] |
| remove(.Z) |

　データセット“.Z”に保存した変数を，データセット“Data1”に追加する．追加後，データセット“.Z”を削除する．

|  |
| --- |
| Data1$close\_ax <- NULL |
| Data1$close\_b <- NULL |
| Data1$close\_issue <- NULL |
| Data1$open\_ax <- NULL |
| Data1$open\_b <- NULL |
| Data1$open\_issue <- NULL |
| Data1$time\_average <- NULL |
| Data1$time\_stdev <- NULL |

　データセット“.Z”から追加された変数が標準化されていることを確認し，データセット“Data1”の標準化前の変数を削除する．

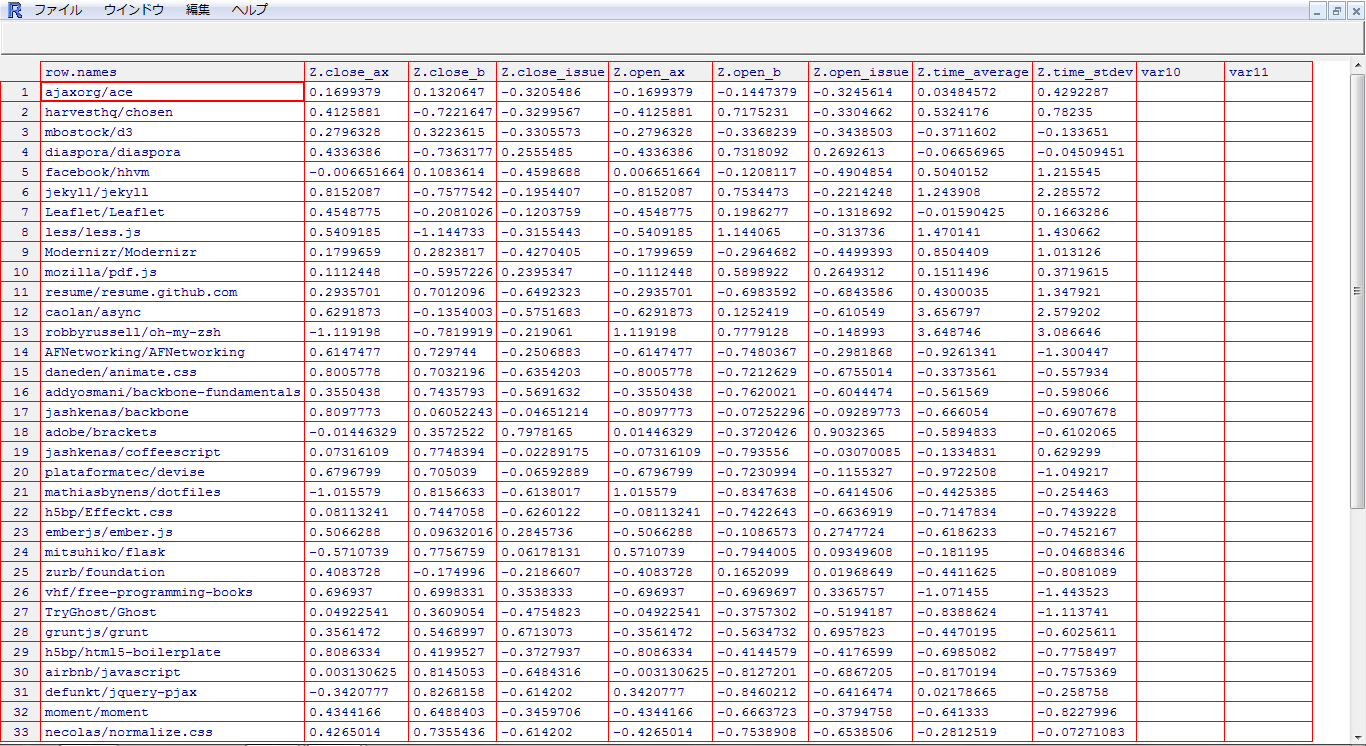


図 15　Data1の表示

　図15はデータセット“Data1”の表示画面である．Star数，Contributors数，Fork数が削除され，変数が全て標準化されているかを確認する．

1. 階層クラスター分析

|  |
| --- |
| HClust.1 <- hclust(dist(model.matrix(~-1 + Z.close\_ax+Z.close\_b+Z.close\_issue+Z.open\_ax+Z.open\_b+Z.open\_issue+Z.time\_average+Z.time\_stdev, Data1))^2 , method= "ward") |

　クラスタリングの方法をウォード法とし，距離の測度をユークリッド距離の平方とする．作成したクラスタリング法をデータセット“HClust.1”に保存する．

|  |
| --- |
| plot(HClust.1, main= "Cluster Dendrogram for Solution HClust.1", xlab= "Observation Number in Data Set Data1", sub="Method=ward; Distance=squared-euclidian") |

　データセット“HClust.1”のデンドログラムを描画する．

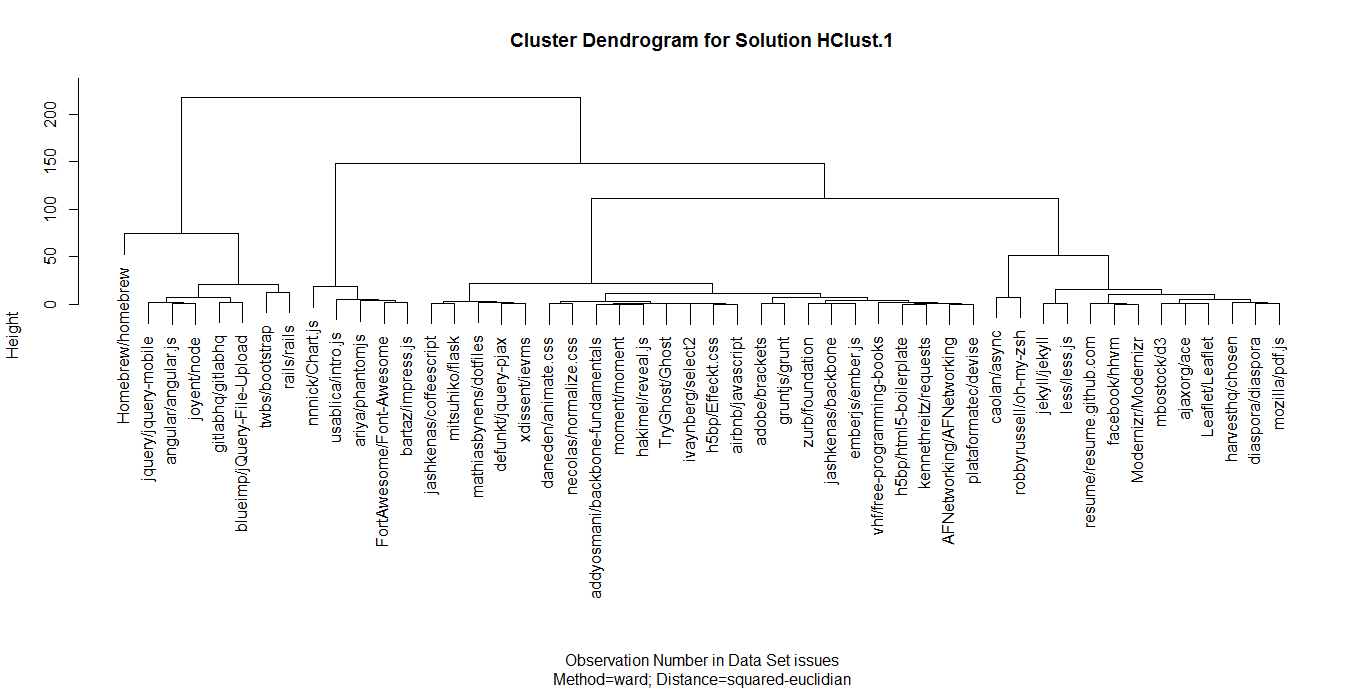


図 16　階層クラスター分析結果

図16はStar数ランキング上位50件のプロジェクトを階層クラスター分析した結果である．クラスター数を多く設定すると，似通った内容のクラスターラベルが発生してしまう．そのため，クラスター数を4つとする．

1. サマリの表示

|  |
| --- |
| summary(as.factor(cutree(HClust.1, k = 4))) # Cluster Sizes |

　クラスター数を4つとし，データセット“HClust.1”のサマリを表示する．

表 10　 HClust.1のサマリ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| クラスターラベル | 1 | 2 | 3 | 4 |
| プロジェクト数 | 13 | 24 | 8 | 5 |

　表10は各クラスターラベルに割り当てられているプロジェクトの数を表したものである．

1. 非階層クラスター分析

|  |
| --- |
| by(model.matrix(~-1 + Z.close\_ax + Z.close\_b + Z.close\_issue + Z.open\_ax + Z.open\_b + Z.open\_issue + Z.time\_average + Z.time\_stdev, Data1), as.factor(cutree(HClust.1, k = 4)),  colMeans) # Cluster Centroids |

　クラスター数を4つとし，データセット“HClust.1”を非階層クラスター分析する．

表 11 　HClust.1の主成分表

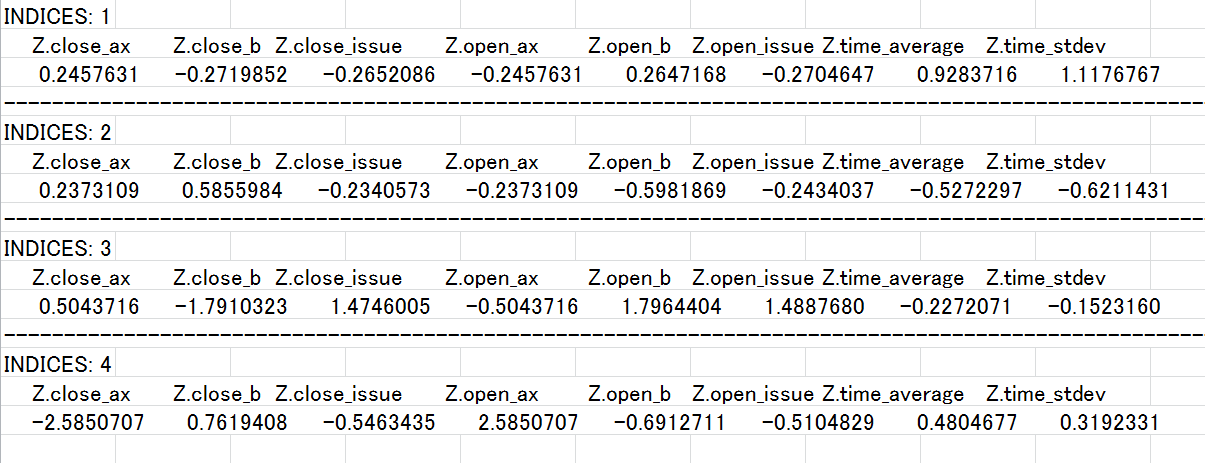


　表11はプロジェクトを4つのパターンに分類する際に，何を主成分として分類したのかを記述した表である．数値が最も高い変数，または最も低い変数を主成分としている．

|  |
| --- |
| biplot(princomp(model.matrix(~-1 + Z.close\_ax + Z.close\_b + Z.close\_issue + Z.open\_ax + Z.open\_b + Z.open\_issue + Z.time\_average + Z.time\_stdev, Data1)), xlabs =  as.character(cutree(HClust.1, k = 4))) |

　データセット“HClust.1”のバイプロットを描画する．

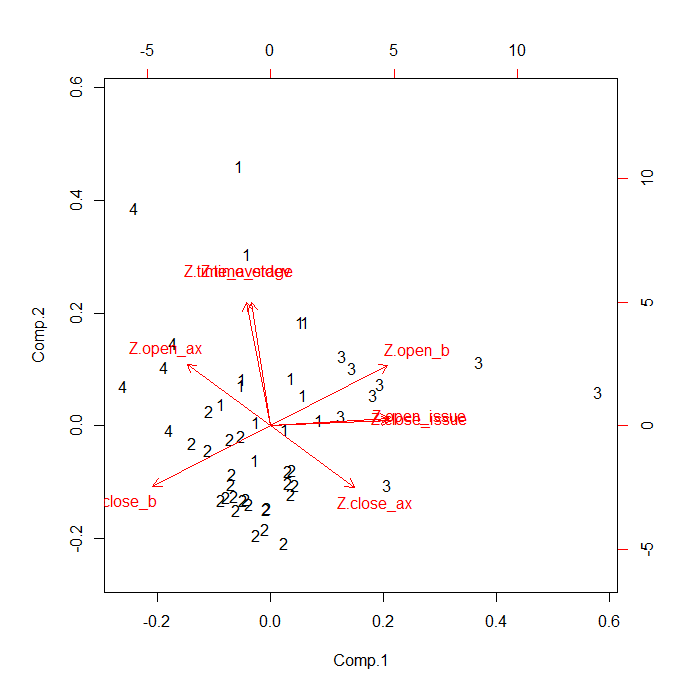


図 17　非階層クラスター分析結果

　図17はStar数ランキング上位50件のプロジェクトを非階層クラスター分析した結果である．クラスター数は4つであり，階層クラスター分析の結果と同じ数である．

1. クラスターラベルの追加

|  |
| --- |
| Data1$hclus.label <- assignCluster(model.matrix(~-1 + Z.close\_ax + Z.close\_b + Z.close\_issue + Z.open\_ax + Z.open\_b + Z.open\_issue + Z.time\_average + Z.time\_stdev, Data1), Data1,  cutree(HClust.1, k = 4)) |

　データセット“HClust.1”にクラスターラベルの数値を追加する．

1. ライブラリの読み込み

|  |
| --- |
| library(kohonen) |

　自己組織化マップを使用するため，ライブラリを読み込む．

1. SOMデータの作成

|  |
| --- |
| Data2 <- read.csv("RankData.csv",header=T,row.names=1) |

　データセットを分けるため，Star数ランキング上位50件のプロジェクトのデータを読み込む．データセット名は“Data2”とする．

|  |
| --- |
| Data2$contributors <- NULL |
| Data2$fork <- NULL |
| Data2$star <- NULL |

　分類に不要な変数であるStar数，Contributors数，Fork数を削除する．

|  |
| --- |
| DataSOM <-som(scale(Data2),grid = somgrid(10,5,"rectangular"),rlen=1000) |

　データセット“Data2”の変数を標準化し，10×5の格子状のマップで，学習回数1000回で，データセット“DataSOM”を作成する．

1. 自己組織化マップの作成

|  |
| --- |
| plot(DataSOM,type="mapping",labels=rownames(Data2),main="ポジショニングマップ") |

　データセット“DataSOM”のポジショニングマップを描画する．

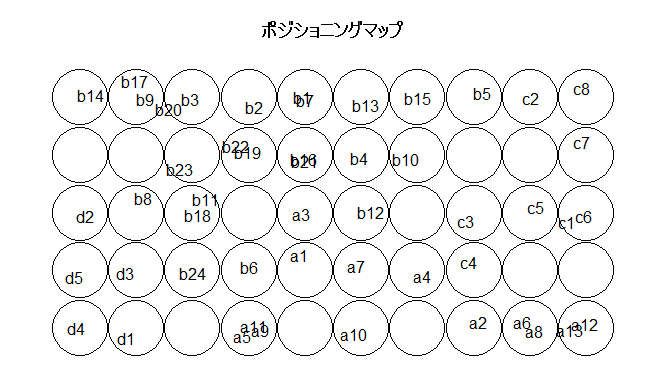


図 18　ポジショニングマップ

　図18はデータセット“DataSOM”のポジショニングマップである．類似度の高いプロジェクトが同じ格子，近くの格子に入る．クラスター分析の結果と同じく4つのパターンに分類できる．

1. 各ユニットのオブザベーション数の作成

|  |
| --- |
| plot(DataSOM,type="counts",main="各ユニットのオブザベーション数") |

データセット“DataSOM”の各ユニットのオブザベーション数を描画する．

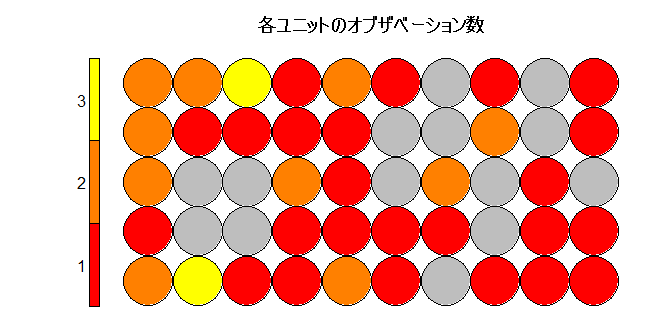


図 19　各ユニットのオブザベーション数

　図19はデータセット“DataSOM”の各ユニットのオブザベーション数である．10×5の格子ごとに，類似しているプロジェクトが幾つあるのかを表している．

1. コード情報の作成

|  |
| --- |
| plot(DataSOM,type="codes",main="コード情報") |

データセット“DataSOM”のコード情報を描画する．



図 20　コード情報

　図20はデータセット“DataSOM”のコード情報である．10×5の格子ごとに，その格子の特徴を表している．

1. 類似度の変化の作成

|  |
| --- |
| plot(DataSOM,type="changes",main="類似度の変化") |

データセット“DataSOM”の類似度の変化を描画する．

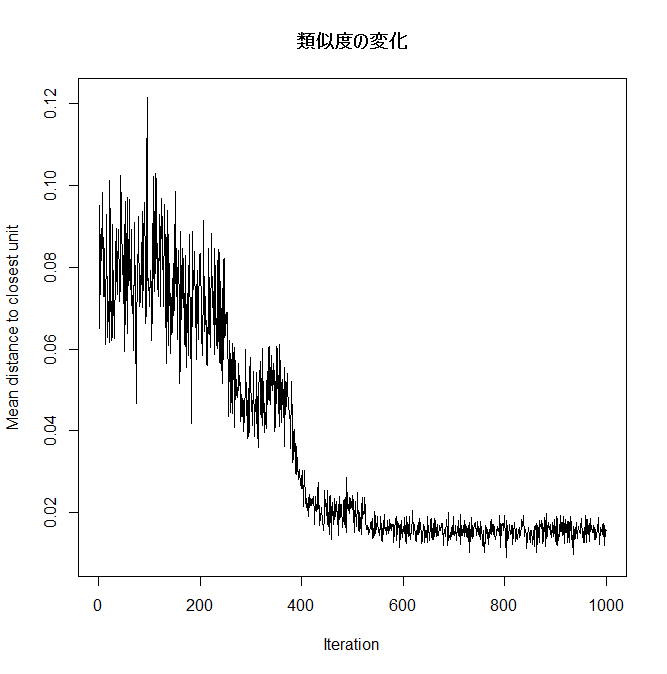


図 21　類似度の変化

　図21はデータセット“DataSOM”の類似度の変化である．学習回数1000回での，データの収束具合を表している．

* + 1. **重回帰分析**

プロジェクトの成功要因であるStar数と関係のある変数を調査するため，重回帰分析をする．

1. データの読み込み

|  |
| --- |
| Data3 <- read.csv("RankData.csv",header=T,row.names=1) |

　データセットを分けるため，Star数ランキング上位50件のプロジェクトのデータを読み込む．データセット名は“Data3”とする．

1. 重回帰分析の実行

|  |
| --- |
| MyModel <- lm(formula = star~fork+contributors+time\_average+time\_stdev+open\_issue+open\_ax+open\_b+close\_issue+close\_ax+close\_b, data = Data3) |

　Star数を目的変数とし，その他の変数を説明変数とする．結果はデータセット“MyModel”に保存する．

1. サマリの表示

|  |
| --- |
| summary(MyModel) |

　データセット“MyModel”のサマリを表示する．

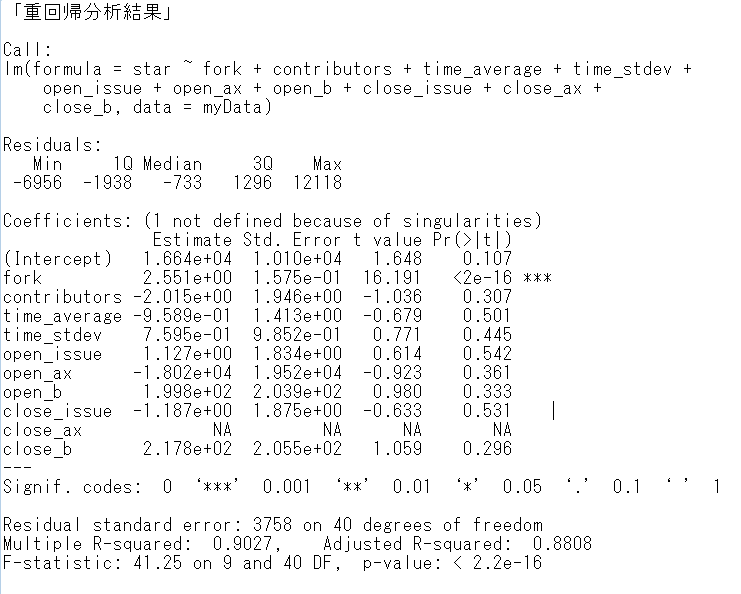


図 22　重回帰分析結果

　図22はデータセット“MyModel”の重回帰分析結果である．Star数と関係のある変数はFork数のみである．

1. 変数の選択

|  |
| --- |
| MyModel2 <- step(MyModel) |

　データセット“MyModel”からStar数と関係が薄い変数を削除し，データセット“MyModel2”に保存する．

|  |
| --- |
| summary(MyModel2) |

　データセット“MyModel2”のサマリを表示する．

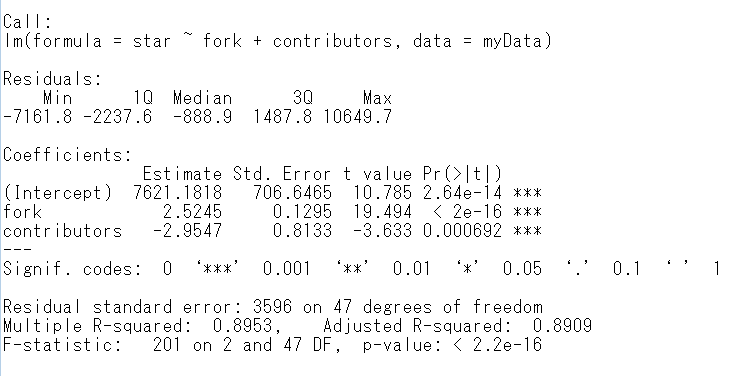


図 23　変数選択後の重回帰分析結果

　図23はデータセット“MyModel2”の重回帰分析結果である． Star数と関係のある変数はFork数，Contributors数である．得られた回帰式はy = 2.5245x1 - 2.9547x2 + 7621.1818である．Star数を求めるには，x1にFork数を代入し，x2にContributors数を代入する．

1. 相関関係図の作成

|  |
| --- |
| plot(MyModel2, panel = panel.smooth) |

　データセット“MyModel”の変数の相関関係図を描画する．

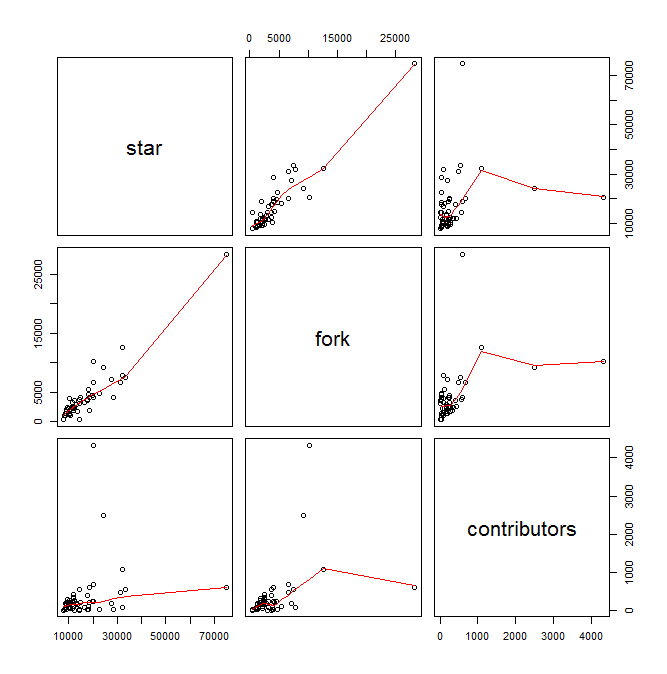


図 24　MyModel2の相関関係図

　図24はデータセット“MyModel2”の相関関係図である．目的変数をStar数とし，説明変数をFork数，Contributors数とする．結果から，Fork数とContributors数が逆相関の関係にあることが分かる．

* + 1. **プログラムの開発**

　以下に本研究でIssuesを取得するために開発したプログラムを記述する．

1. api.py

|  |
| --- |
| #!/usr/bin/python  # coding: UTF-8  import sys, json, requests  #GitHubのログイン情報をファイルから取得する  #TODO:パスワードに「:」を使っているとダメ  tmp = open(‘github.passwd’).readline().rstrip(‘\n’).split(‘:’);  username = tmp[0]  password = tmp[1]  #print >> sys.stderr, username,password  #APIのURLはコマンドライン引数で与える  url = sys.argv[1]  count = 0  while (url is not None):  print >> sys.stderr, url  r = requests.get(url, auth=(username, password))  print >> sys.stderr, r.headers[‘status’],  items = r.json()[‘items’] if ‘items’ in r.json() else r.json()  for item in items:  count = count + 1  print json.dumps(item)  if (r.links.has\_key(‘next’)):  url = r.links[‘next’][‘url’]  else:  url = None  print >> sys.stderr, count, ‘items’ |

　Python形式で開発したプログラムである．GitHubログイン情報入力の省略，および各種設定を行う．

1. Issues.sh

|  |
| --- |
| #!/bin/bash  python api.py "https://api.github.com/repos/ユーザ名/リポジトリ名/issues?per\_page=100" > ユーザ名-リポジトリ名-openissues.txt  python api.py "https://api.github.com/repos/ユーザ名/リポジトリ名/issues?per\_page=100&state=closed" > ユーザ名-リポジトリ名-closedissues.txt  ./jq '.created\_at' ユーザ名-リポジトリ名-openissues.txt | awk '{ printf("%s open\n", $0); }' > ユーザ名-リポジトリ名-issues.tmp  ./jq '.created\_at,.closed\_at' ユーザ名-リポジトリ名-closedissues.txt | awk '{ if (NR % 2 == 1) printf("%s open\n", $0); else printf("%s close\n", $0); }' >> ユーザ名-リポジトリ名-issues.tmp  sort ユーザ名-リポジトリ名-issues.tmp > ユーザ名-リポジトリ名-issues.txt  awk 'BEGIN { openissues=0; closedissues=0; } $2=="open" { openissues++; } $2=="close" { closedissues++; } { printf("%s,%d,%d\n", $1, openissues, closedissues) }' ユーザ名-リポジトリ名-issues.txt > ユーザ名-リポジトリ名-issues.csv |

　指定したプロジェクトからOpenIssuesとClosedIssuesを取得するプログラムである．

第5章

結果・考察

# **結果・考察**

* 1. **本章の構成**

　本章では，4章で記述した調査方法を用いて取得した調査結果を記述する．また，調査結果に対する考察も記述する．

* 1. **調査結果**

　Star数ランキング上位50件のプロジェクトとランダムに選択した50件のプロジェクトの調査結果を記述する．

* + 1. **プロジェクト成功の成否の調査結果**

　以下にStar数ランキング上位50件のプロジェクトとランダムに選択した50件のプロジェクトの成功数と失敗数を記述する．成功・失敗の判断はリリースしているソフトウェアのバージョンから行う．Ver. 1.0以上をリリースしているプロジェクトを成功とし，Ver. 1.0未満でリリースが止まっているプロジェクトを失敗とする．

表 12　Star数ランキング上位50件のプロジェクトの成否表

|  |  |  |
| --- | --- | --- |
| Star数＼リリースバージョン | ver. 1.0 未満 (失敗) | ver. 1.0 以上 (成功) |
| 0 ~ 10000 | 4 | 6 |
| 10001 ~ 20000 | 4 | 24 |
| 20001 ~ 30000 | 1 | 6 |
| 30001 ~ | 1 | 4 |
| 合計プロジェクト件数 | 10 | 40 |

　表12はStar数ランキング上位50件のプロジェクトの成否表である．成功しているプロジェクトは50件中40件であり，失敗しているプロジェクトは50件中10件である．

表 13　ランダムに選択した50件のプロジェクトの成否表

|  |  |  |
| --- | --- | --- |
| Star数＼リリースバージョン | ver. 1.0 未満 (失敗) | ver. 1.0 以上 (成功) |
| 0 ~ 500 | 3 | 3 |
| 501~1000 | 9 | 13 |
| 1001~1500 | 3 | 4 |
| 1501~ | 5 | 10 |
| 合計プロジェクト件数 | 20 | 30 |

　表13はランダムに選択した50件のプロジェクトの成否表である．成功しているプロジェクトは50件中30件であり，失敗しているプロジェクトは50件中20件である．

　Star数が増加すると，成功しているプロジェクトが増加し，失敗しているプロジェクトが減少する．Star数が減少すると，成功しているプロジェクトが減少し，失敗しているプロジェクトが増加する．この結果から，Star数がプロジェクトの成功要因であることを証明できる．

* + 1. **Star数ランキング上位50件のプロジェクトの調査結果**

　調査には，階層クラスター分析，非階層クラスター分析，自己組織化マップ，重回帰分析を使用した．以下にStar数ランキング上位50件のプロジェクトの調査結果を記述する．

1. 階層クラスター分析結果

　以下に階層クラスター分析の結果と各クラスターラベルの特徴を記述する．

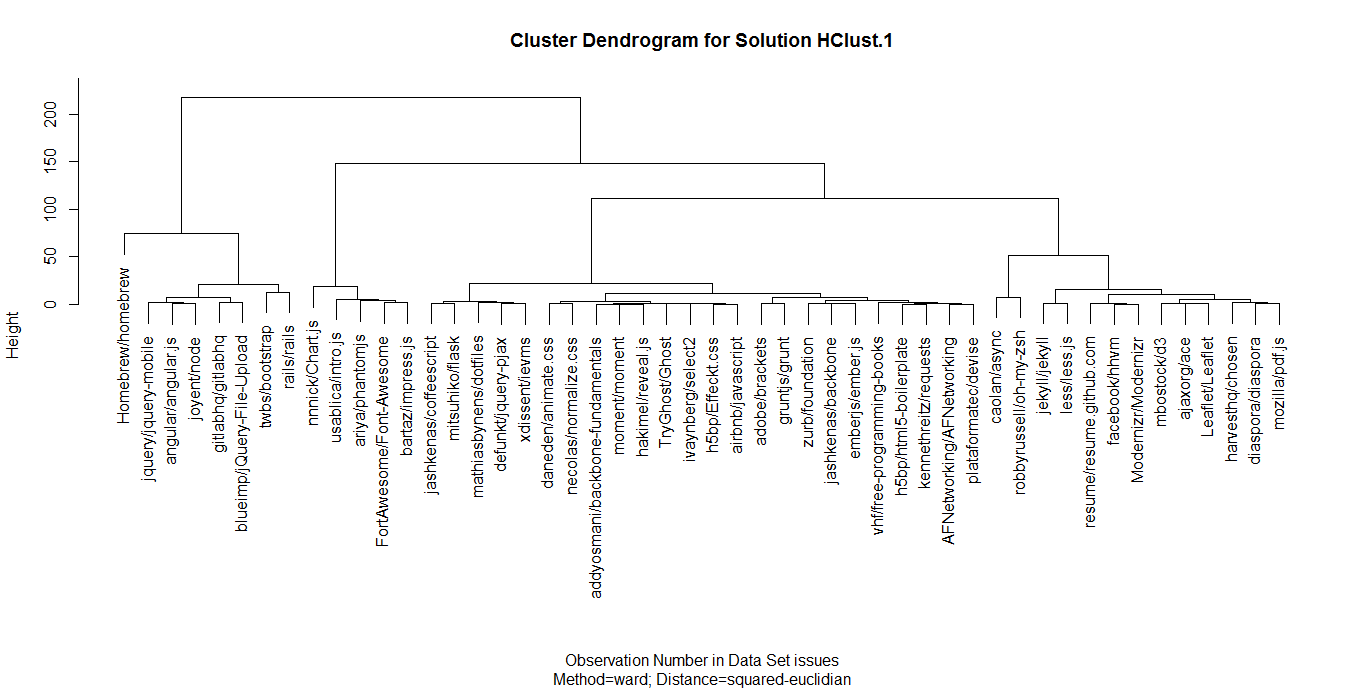


図 25　階層クラスター分析結果　上位50件

　図25はStar数ランキング上位50件のプロジェクトの階層クラスター分析結果である．クラスター数を4つとし，プロジェクトを4つのパターンに分類する．以下に各クラスターラベルの特徴を記述する．

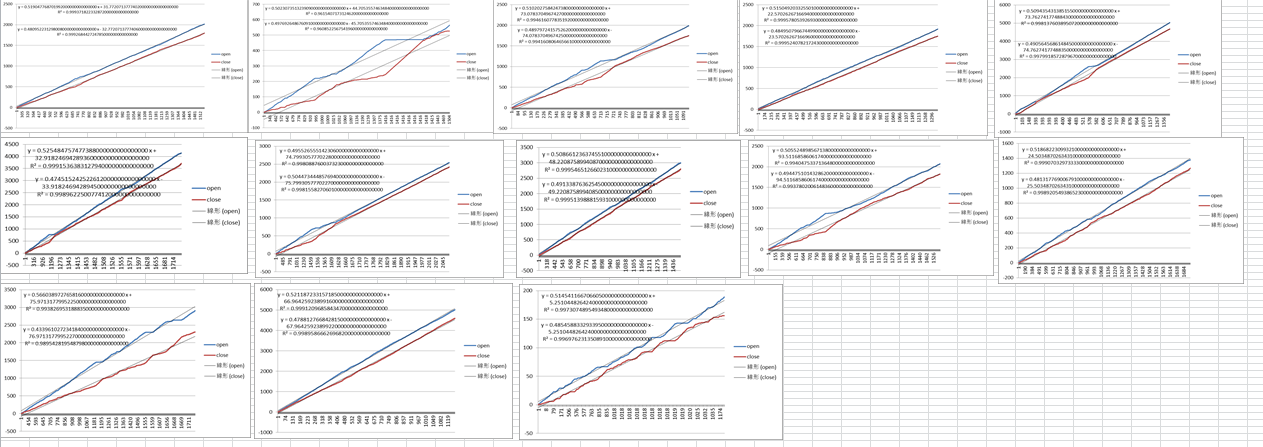


図 26　クラスターラベル1　上位50件

　図26はクラスターラベル1に属するプロジェクトのグラフである．これらのプロジェクトの特徴として，Issuesを完了するまでの所要時間の平均と標準偏差の値が大きいことが挙げられる．

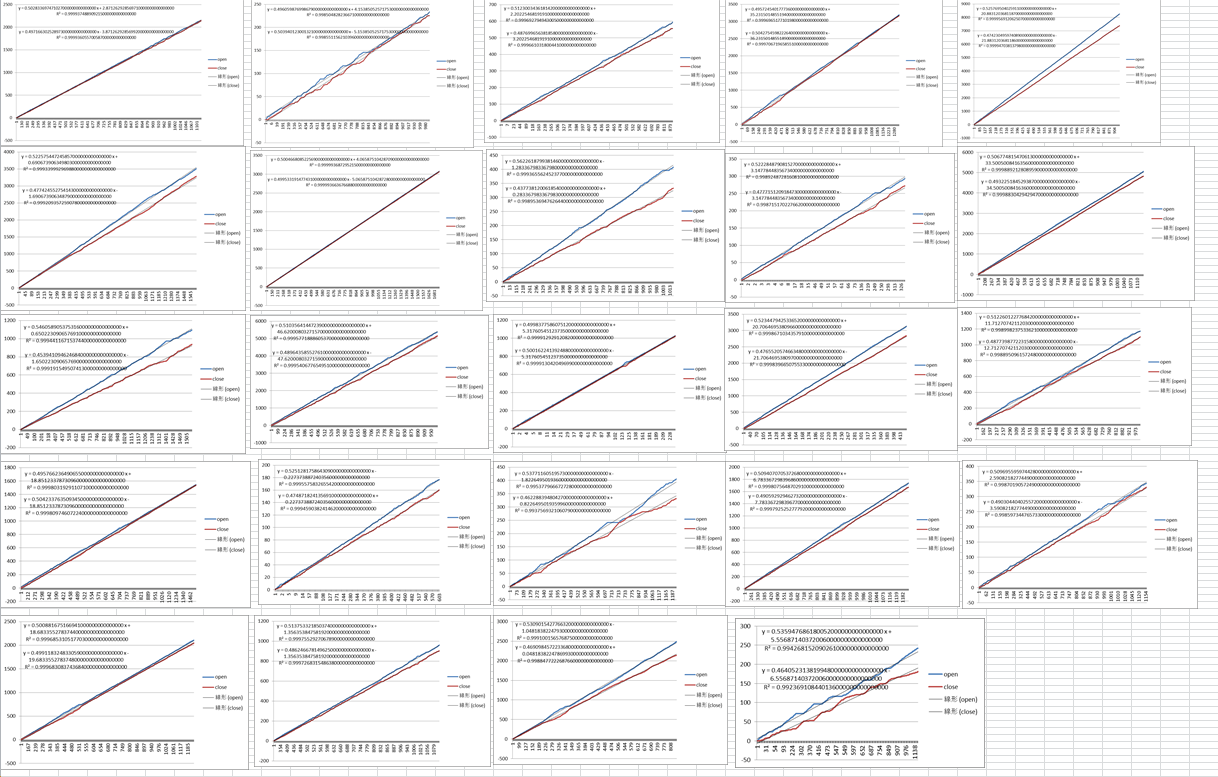


図 27　クラスターラベル2　上位50件

図27はクラスターラベル2に属するプロジェクトのグラフである．4つのクラスターラベルの中で最もプロジェクト成功数が多い．これらのプロジェクトの特徴として，Issuesを完了するまでの所要時間の平均と標準偏差の値が小さいこと，OpenIssuesとClosedIssuesの切片の絶対値が小さいことが挙げられる．

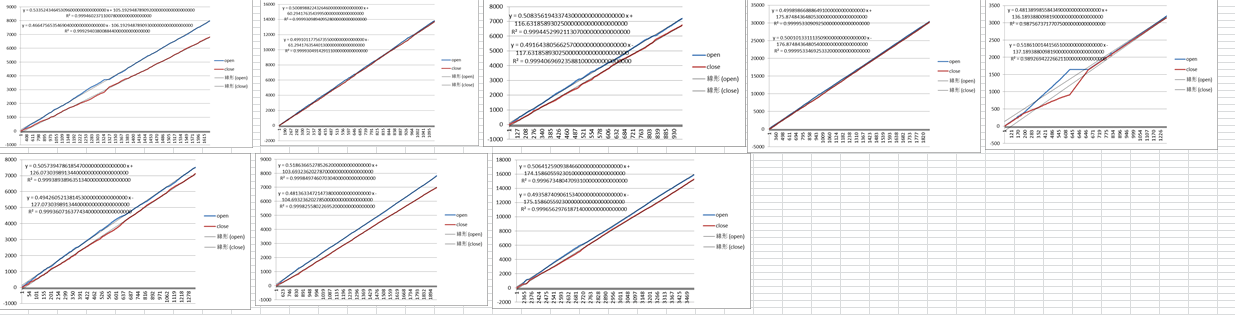


図 28　クラスターラベル3　上位50件

図28はクラスターラベル3に属するプロジェクトのグラフである．これらのプロジェクトの特徴として，OpenIssues数とClosedIssues数が多いこと，OpenIssuesとClosedIssuesの切片の絶対値が大きいことが挙げられる．

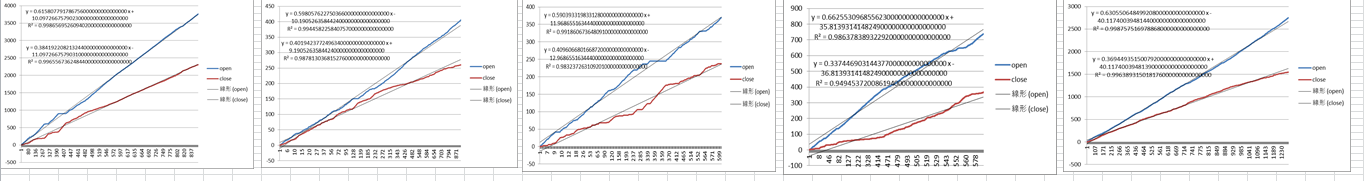


図 29　クラスターラベル4　上位50件

図29はクラスターラベル4に属するプロジェクトのグラフである．4つのクラスターラベルの中で最もプロジェクト失敗数が多い．これらのプロジェクトの特徴として，Issuesを完了するまでの所要時間の平均と標準偏差の値が大きいこと，OpenIssues数とClosedIssues数が少ないこと，OpenIssuesの傾きの値が大きくClosedIssuesの傾きの値が小さいことが挙げられる．

1. 非階層クラスター分析結果

　以下に非階層クラスター分析の結果を記述する．

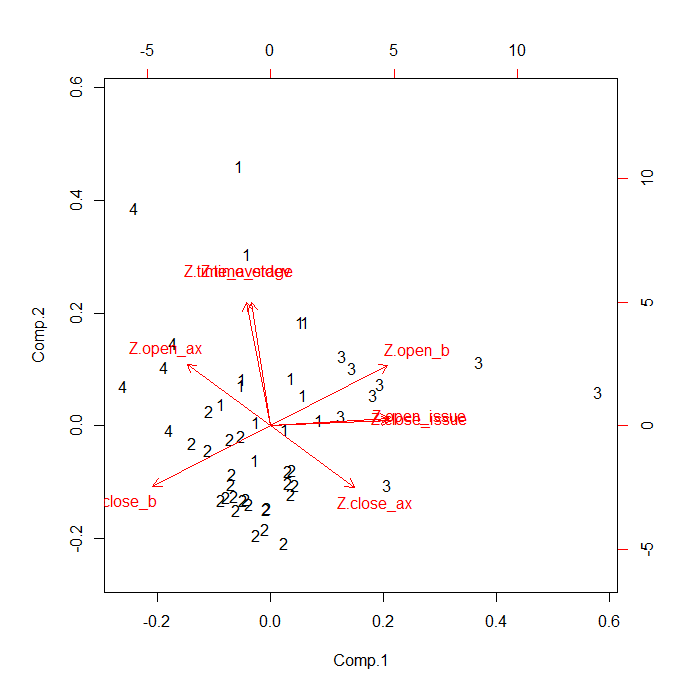


図 30　非階層クラスター分析結果　上位50件

　図30はStar数ランキング上位50件のプロジェクトの非階層クラスター分析結果である．クラスター数を4つとし，プロジェクトを4つのパターンに分類する．表示されている数字はプロジェクトである．数字の値はプロジェクトが属しているクラスターラベルを示している．

　図30から，50件のプロジェクトを4つに分類できていることが分かる．

1. 自己組織化マップの作成結果

　以下に自己組織化マップの作成結果を記述する．作成する図はポジショニングマップ，各ユニットのオブザベーション数，コード情報，類似度の変化である．

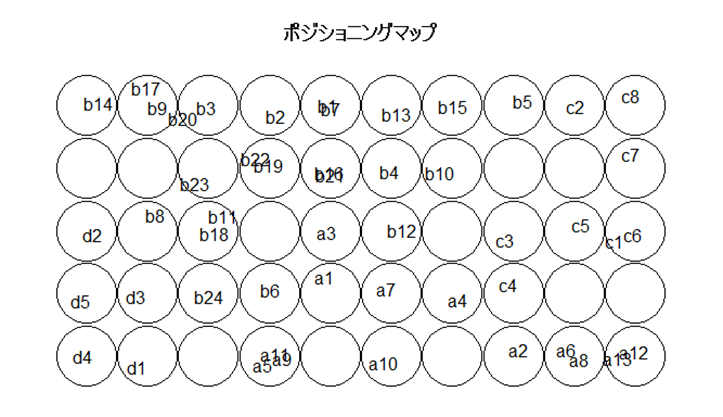


図 31　ポジショニングマップ　上位50件

　図31はStar数ランキング上位50件のプロジェクトのポジショニングマップである．表示されている英数字はプロジェクトである．英字はクラスターラベルを示し，数字はクラスターラベルに属しているプロジェクトの数を示している．a＝クラスターラベル1，b＝クラスターラベル2，c＝クラスターラベル3，d＝クラスターラベル4である．つまり，b13はクラスターラベル2に属している13件目のプロジェクトとなる．

　図31から，50件のプロジェクトを4つに分類できていることが分かる．

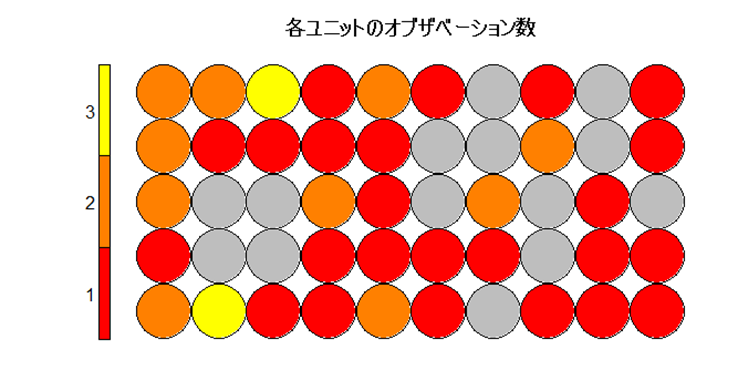


図 32　各ユニットのオブザベーション数　上位50件

　図32はStar数ランキング上位50件のプロジェクトの各ユニットのオブザベーション数である．10×5の格子ごとに，類似しているプロジェクトが幾つあるのかを表している．各ユニットのオブザベーション数は2以上が多いため，類似しているプロジェクトが多いと解釈できる．

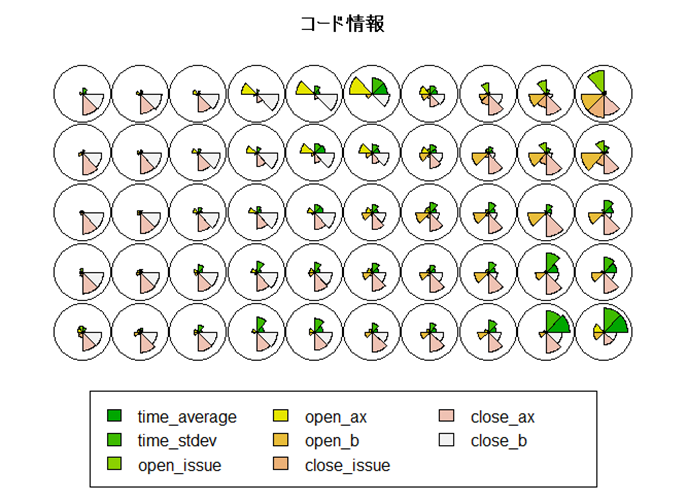


図 33　コード情報　上位50件

　図33はStar数ランキング上位50件のプロジェクトのコード情報である．10×5の格子ごとに，その格子の特徴を表している．階層クラスター分析結果と酷似した結果が得られた．



図 34　類似度の変化　上位50件

　図34はStar数ランキング上位50件のプロジェクトの類似度の変化である．学習回数1000回での，データの収束具合を表している．学習回数400回程でデータが収束している．

1. 重回帰分析結果

　以下に重回帰分析の結果と相関関係図を記述する．

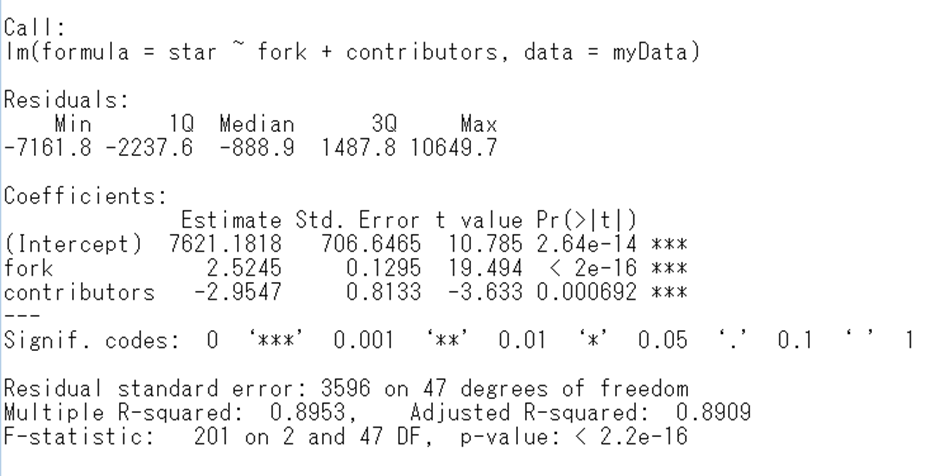


図 35　重回帰分析結果　上位50件

　図35はStar数ランキング上位50件のプロジェクトの重回帰分析結果である．目的変数をStar数とし，説明変数をFork数，Contributors数とする．結果から，この2つの説明変数はStar数に関係しているということが分かる．この分析結果は，決定係数が0.8953と高く，有意であるといえる．

　回帰式はy = 2.5245x1 - 2.9547x2 + 7621.1818であり，Fork数とContributors数は逆相関の関係にある．

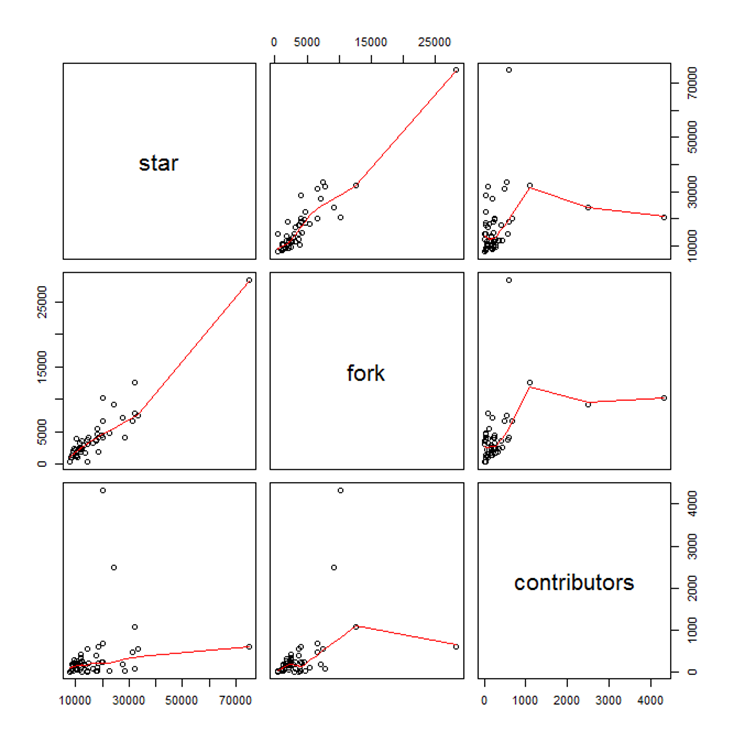


図 36　相関関係図　上位50件

　図36はStar数ランキング上位50件のプロジェクトの相関関係図である．結果から，Fork数とContributors数が逆相関の関係にあることが分かる．これにより，Star数が多いプロジェクトはFork数が多く，Contributors数が少ないということが分かる．

* + 1. **ランダムに選択した50件のプロジェクトの調査結果**

調査には，階層クラスター分析，非階層クラスター分析，自己組織化マップ，重回帰分析を使用した．以下にランダムに選択した50件のプロジェクトの調査結果を記述する．

1. 階層クラスター分析結果

　以下に階層クラスター分析の結果と各クラスターラベルの特徴を記述する．

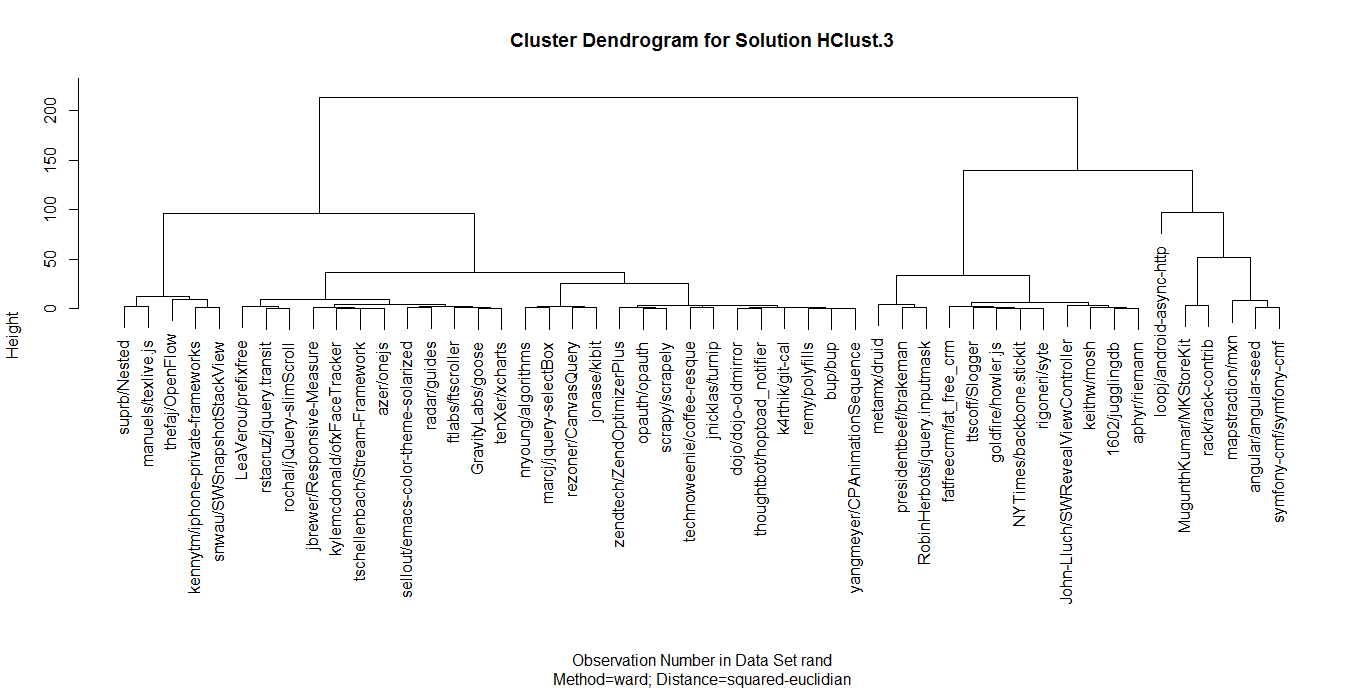


図 37　階層クラスター分析結果　ランダム50件

　図37はランダムに選択した50件のプロジェクトの階層クラスター分析結果である．クラスター数を5つとし，プロジェクトを5つのパターンに分類する．ランダム50件のプロジェクトは，上位50件のプロジェクトと比べてStar数が少なく，OpenIssues数とClosedIssues数も少ない．

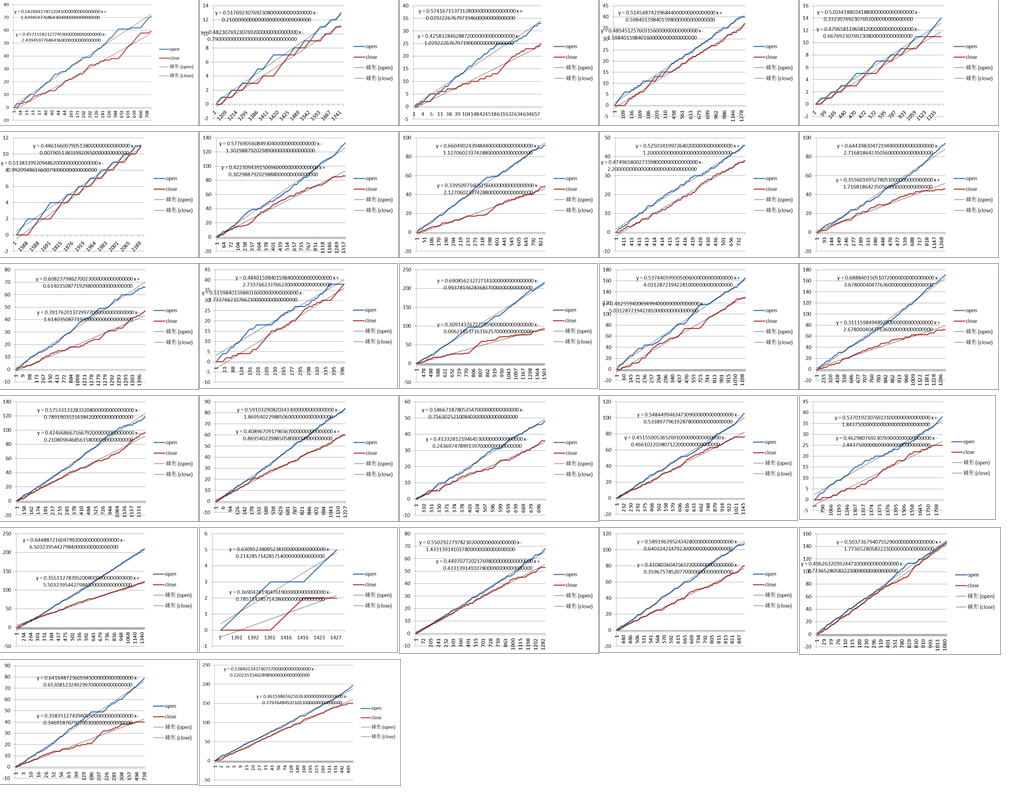


図 38　クラスターラベル1　ランダム50件

図38はクラスターラベル1に属するプロジェクトのグラフである．これらのプロジェクトの特徴として，OpenIssues数とClosedIssues数が少ないことが挙げられる．

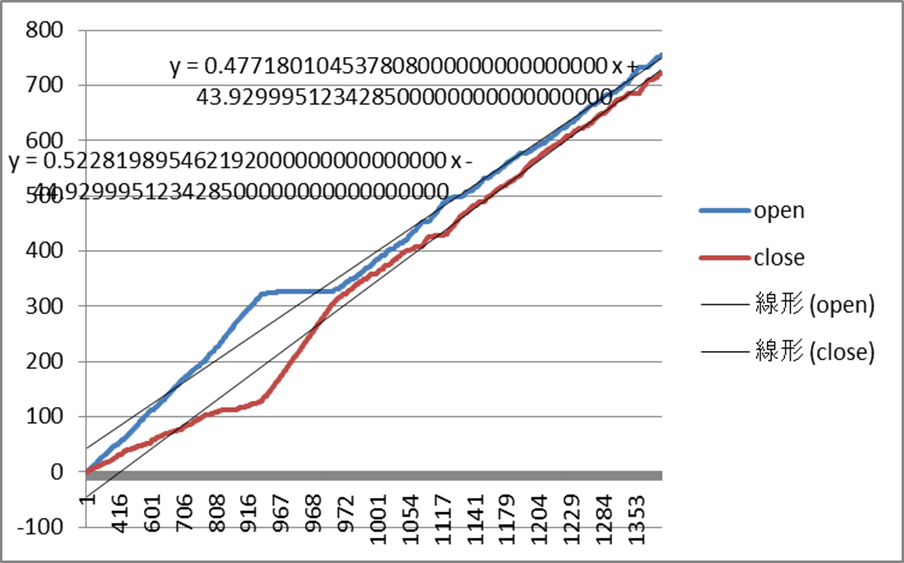


図 39　クラスターラベル2　ランダム50件

図39はクラスターラベル2に属するプロジェクトのグラフである．このプロジェクトの特徴として，OpenIssues数とClosedIssues数が多いこと，OpenIssuesとClosedIssuesの切片の絶対値が大きいことが挙げられる．

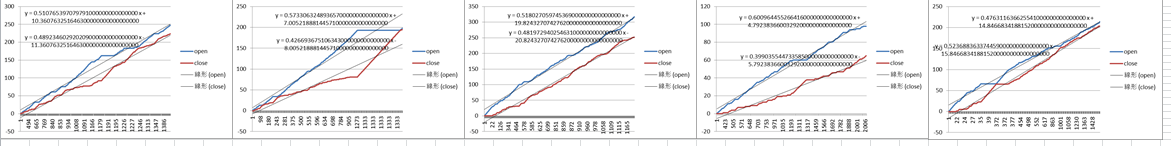


図 40　クラスターラベル3　ランダム50件

図40はクラスターラベル3に属するプロジェクトのグラフである．5つのクラスターラベルの中で最もプロジェクト成功数が多い．これらのプロジェクトの特徴として，Issuesを完了するまでの所要時間の平均と標準偏差の値が大きいこと，OpenIssuesとClosedIssuesの切片の絶対値が大きいことが挙げられる．

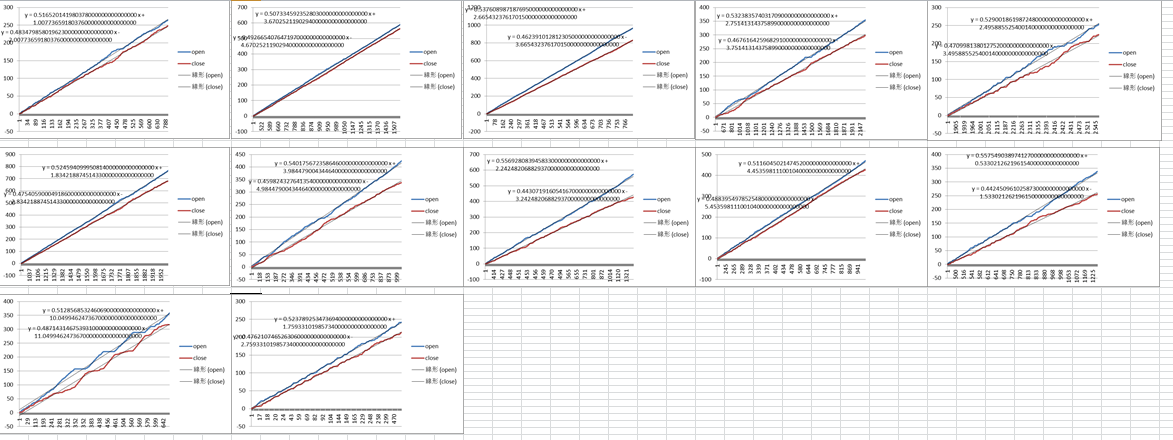


図 41　クラスターラベル4　ランダム50件

図41はクラスターラベル4に属するプロジェクトのグラフである．これらのプロジェクトの特徴として，OpenIssues数とClosedIssues数が多いことが挙げられる．



図 42　クラスターラベル5　ランダム50件

図42はクラスターラベル5に属するプロジェクトのグラフである．5つのクラスターラベルの中で最もプロジェクト失敗数が多い．これらのプロジェクトの特徴として，OpenIssues数とClosedIssues数が少ないこと，OpenIssuesの傾きの値が大きくClosedIssuesの傾きの値が小さいことが挙げられる．

1. 非階層クラスター分析結果

　以下に非階層クラスター分析の結果を記述する．

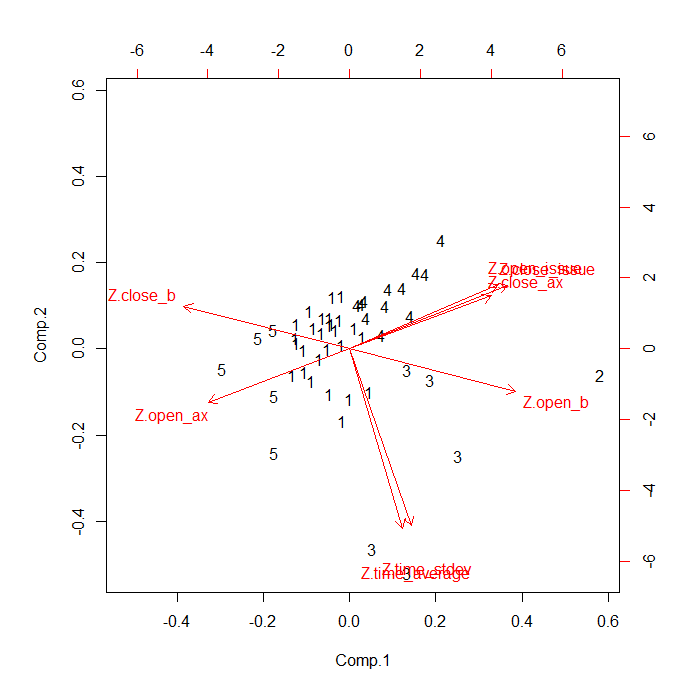


図 43　非階層クラスター分析結果　ランダム50件

　図43はランダムに選択した50件のプロジェクトの非階層クラスター分析結果である．クラスター数を5つとし，プロジェクトを5つのパターンに分類する．表示されている数字はプロジェクトである．数字の値はプロジェクトが属しているクラスターラベルを示している．

　図43から，50件のプロジェクトを5つに分類できていることが分かる．

1. 自己組織化マップの作成結果

　以下に自己組織化マップの作成結果を記述する．作成する図はポジショニングマップ，各ユニットのオブザベーション数，コード情報，類似度の変化である．

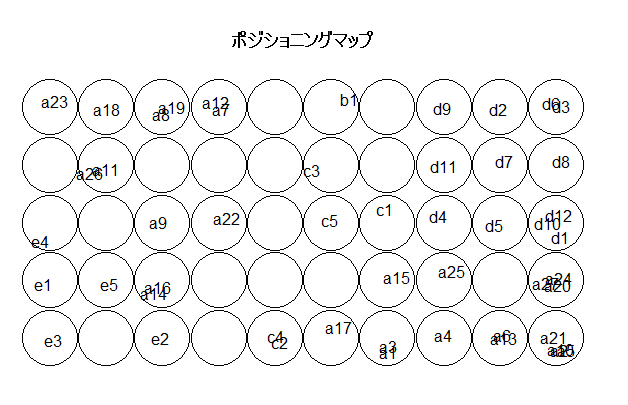


図 44　ポジショニングマップ　ランダム50件

　図44はランダムに選択した50件のプロジェクトのポジショニングマップである．表示されている英数字はプロジェクトである．英字はクラスターラベルを示し，数字はクラスターラベルに属しているプロジェクトの数を示している．a＝クラスターラベル1，b＝クラスターラベル2，c＝クラスターラベル3，d＝クラスターラベル4，e＝クラスターラベル5である．つまり，e8はクラスターラベル5に属している8件目のプロジェクトとなる．

　図44から，50件のプロジェクトを5つに分類できていることが分かる．

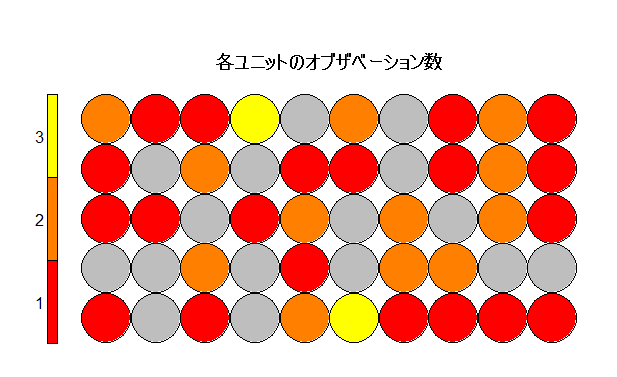


図 45　各ユニットのオブザベーション数　ランダム50件

　図45はランダムに選択した50件のプロジェクトの各ユニットのオブザベーション数である．10×5の格子ごとに，類似しているプロジェクトが幾つあるのかを表している．各ユニットのオブザベーション数は2以上が多いため，類似しているプロジェクトが多いと解釈できる．

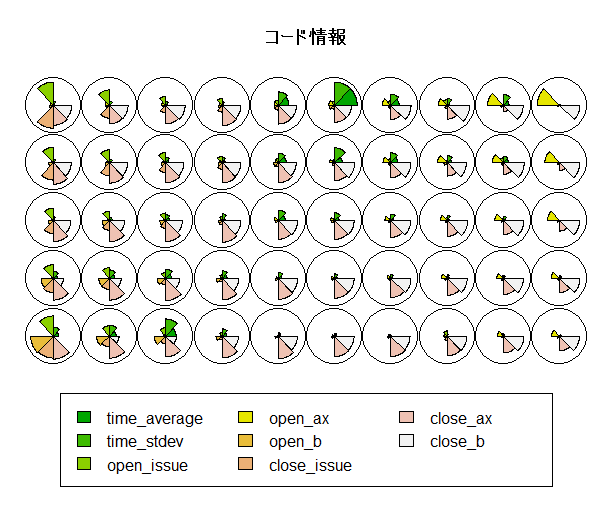


図 46　コード情報　ランダム50件

　図46はランダムに選択した50件のプロジェクトのコード情報である．10×5の格子ごとに，その格子の特徴を表している．階層クラスター分析結果と酷似した結果が得られた．

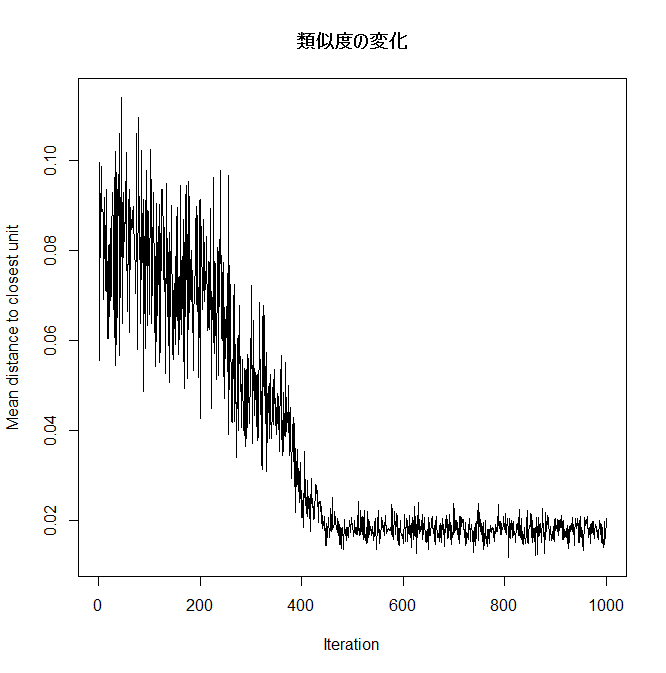


図 47　類似度の変化　ランダム50件

　図47はランダムに選択した50件のプロジェクトの類似度の変化である．学習回数1000回での，データの収束具合を表している．学習回数400回程でデータが収束している．

1. 重回帰分析結果

　以下に重回帰分析の結果と相関関係図を記述する．

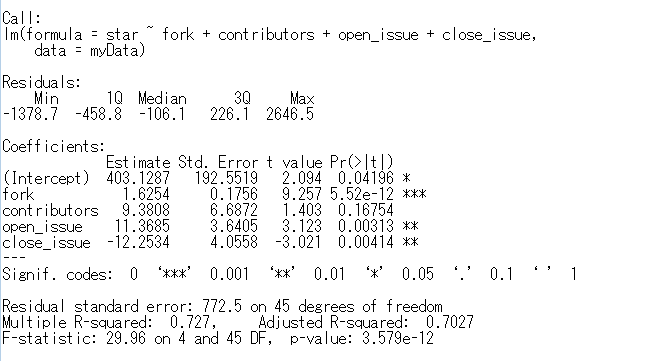


図 48　重回帰分析　ランダム50件

　図48はランダムに選択した50件のプロジェクトの重回帰分析結果である．目的変数をStar数とし，説明変数をFork数，Contributors数，OpenIssues数，ClosedIssues数とする．結果から，Contributors数を除いた3つの説明変数はStar数に関係しているということが分かる．この分析結果は，決定係数が0.727であり，有意であるといえる．

　回帰式はy = 1.6254x1 + 9.3808x2 + 11.3685x3 - 12.2534x4 + 403.1287であり，ClosedIssues数はFork数，Contributors数，OpenIssues数と逆相関の関係にある．

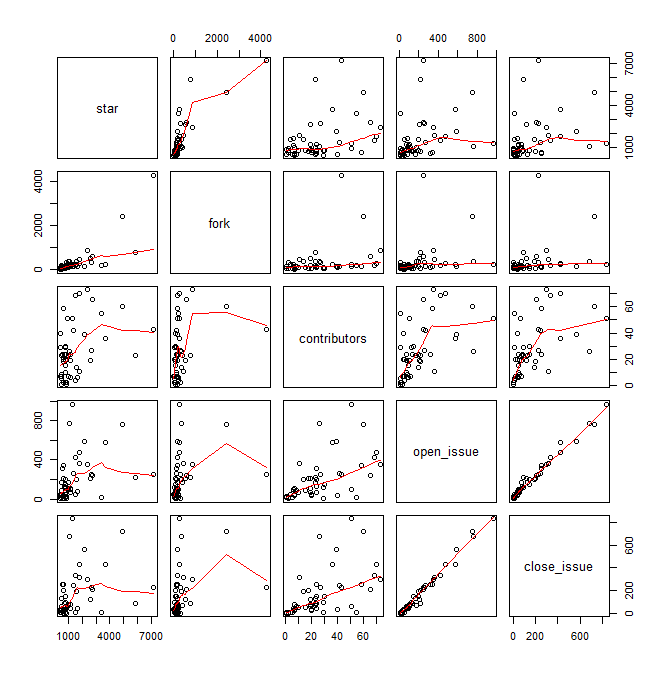


図 49　相関関係図　ランダム50件

　図49はランダムに選択した50件のプロジェクトの相関関係図である．結果から，ClosedIssues数はFork数，Contributors数，OpenIssues数と逆相関の関係にあることが分かる．これにより， Star数が多いプロジェクトはFork数，Contributors数，OpenIssues数が多く，ClosedIssues数が少ないということが分かる．

* 1. 調査結果まとめ

　Star数ランキング上位50件のプロジェクトとランダムに選択した50件のプロジェクトを時系列解析し，プロジェクト成功の成否に関連するパターンを発見した．成功しているプロジェクトの共通点として，Star数が多いこと，Issuesを完了するまでの所要時間が短いこと，Issuesの切片の絶対値が小さいことが挙げられる．また，失敗しているプロジェクトの共通点として，Star数が少ないこと，Issuesを完了するまでの所要時間が長いこと，OpenIssuesの傾きの値が大きくClosedIssuesの傾きの値が小さいこと，OpenIssues数とClosedIssues数が少ないことが挙げられる．

　以上の結果から，「Star数が多く，Issuesを完了するまでの所要時間が短く，Issuesの切片の絶対値が小さいパターン」が，プロジェクト成功の成否に関連するパターンである．

* 1. **考察**

　Star数ランキング上位50件のプロジェクトとランダムに選択した50件のプロジェクトの調査結果から考察を行う．

1. プロジェクト成功の成否についての考察

　成功しているプロジェクトの共通点として，Star数が多いこと，Issuesを完了するまでの所要時間が短いこと，OpenIssuesとClosedIssuesの切片の絶対値が小さいことが挙げられる．また，失敗しているプロジェクトの共通点として，Star数が少ないこと，Issuesを完了するまでの所要時間が長いこと，OpenIssuesの傾きの値が大きくClosedIssuesの傾きの値が小さいこと，OpenIssues数とClosedIssues数が少ないことが挙げられる．これら共通点がプロジェクト成功の成否に関連する理由は，プロジェクトの進捗に関わるからであると考えられる．

　Star数がプロジェクトの進捗に関わる理由として，プロジェクトの注目度との関係性が挙げられる．Star数の多いプロジェクトは人の目に付きやすく，注目度が高くなる．注目度が高いと，プロジェクトがForkされやすくなり，Pull Requestによる他者からの協力が期待できるようになる．この点で，Star数がプロジェクトの進捗に関わっていると考えられる．

　Issuesを完了するまでの所要時間がプロジェクトの進捗に関わる理由として，Issuesの完了速度との関係性が挙げられる．Issuesは作業指示書であり，Issuesの完了は1つの作業の完了を表す．つまり，Issuesを完了するまでの所要時間が短ければプロジェクトの進捗が良くなり，長ければ進捗は悪くなる．この点で，Issuesを完了するまでの所要時間がプロジェクトの進捗に関わっていると考えられる．

　Issuesの線形式の切片の絶対値がプロジェクトの進捗に関わる理由として，Issuesの進捗状況との関係性が挙げられる．OpenIssuesとClosedIssuesの切片の絶対値が大きければ，Issuesの完了速度が速い．これは，進捗状況が良好であることを表している．一方，OpenIssuesとClosedIssuesの切片の絶対値が小さければ，Issuesの発行速度と完了速度は釣り合いが取れている．これは，進捗状況が安定していることを表している．切片の絶対値は，Issuesの進捗状況を数値化したものである．この点で，Issuesの線形式の切片の絶対値がプロジェクトの進捗に関わっていると考えられる．

　Issuesの線形式の傾きの値がプロジェクトの進捗に関わる理由として，Issuesの進捗状況との関係性が挙げられる． OpenIssuesとClosedIssuesの傾きの値が同等であれば，Issuesの発行速度と完了速度は釣り合いが取れている．これは，進捗状況が良好であることを表している．一方，OpenIssuesの傾きの値が大きく，ClosedIssuesの傾きの値が小さければ，Issuesの発行速度が完了速度を上回っている．これは，進捗状況が良好でないことを表している．傾きの値は，Issuesの進捗状況を数値化したものである．この点で，Issuesの線形式の傾きの値がプロジェクトの進捗に関わっていると考えられる．

　Issues数がプロジェクトの進捗に関わる理由として，ソフトウェアの品質との関係性が挙げられる．Issues数が多いことは，機能の追加やバグの修正が頻繁に行われていることを表している．つまり， Issues数の多さはソフトウェアの品質の高さに繋がっている．品質の高いソフトウェアを開発すれば，プロジェクトの注目度が高くなり，Star数が増加する．Star数が多いとプロジェクトの進捗は良くなる．この点で，Issues数がプロジェクトの進捗に関わっていると考えられる．

1. 重回帰分析の考察

　Star数に関係している変数は，Star数ランキング上位50件のプロジェクトとランダムに選択した50件のプロジェクトでは異なっている．その理由として，プロジェクトの規模の違いが挙げられる．Star数ランキング上位50件のプロジェクトは，プロジェクトの規模が大きく，Fork数，Contributors数，Issues数も多い．一方，ランダムに選択した50件のプロジェクトは，プロジェクトの規模が小さく，Fork数，Contributors数，Issues数も少ない．2種類のプロジェクト群の変数の値には大きな差があるため，変数選択に影響が出てしまっていると考えられる．

　ランダムに選択した50件のプロジェクトの回帰式にOpenIssues数とClosedIssues数が含まれている理由は，決定係数を上昇させるためであると考えられる．そのため，信頼性が高い回帰式はStar数ランキング上位50件のプロジェクトの方であるだろう．

1. 考察まとめ

　機能の追加やバグ修正が頻繁に発生するOSS開発プロジェクトにおいて，Issuesをいかに早く完了できるかは重要である．本研究で「Star数が多く，Issuesを完了するまでの所要時間が短く，Issuesの切片の絶対値が小さいパターン」がプロジェクト成功の成否に関連するパターンであるという調査結果が出たが，これは正しい結果であると考えられる．

# **参考文献**

1. Ingrid Lunden. “GitHub Hits The 4M User Mark As It Looks Beyond Developers For Its Next Stage Of Growth”. TechCrunch. 2013-9-7. http://techcrunch.com/2013/09/11/github-hits-the-4m-user-mark-as-it-looks-beyond-developers-for-its-next-stage-of-growth/, (参照 2014-09-01).
2. Briandoll. “10 Million Repositories”. GitHub. 2013-12-23. https://github.com/blog/1724-10-million-repositories, (参照 2014-09-01).
3. 小川明彦, 酒井誠. チケット駆動開発. 翔泳社, 2012-8-23.
4. 久保孝樹. チケットを活用するオープンソースソフトウェア開発の実態調査. 千葉工業大学, 2013, 卒業論文.
5. Project Management Institute, Inc. プロジェクトマネジメント知識体系ガイド (第4版). PMI, 2009-12.
6. 天沼健仁, TIS株式会社. “ガチで5分で分かるITS/BTS&使えるツール6選 (3/7)”. @IT. 2013-06-27. http://www.atmarkit.co.jp/ait/articles/1306/26/news012\_3.html, (参照 2014-09-26).
7. Sean Osawa. “チケット駆動開発を上手に運用するためのプラクティス (ゲストブログ)”. Atlassian Blogs. 2012-10-02. http://japan.blogs.atlassian.com/2012/10/tidd-part2/, (参照 2014-09-26).
8. “チケット駆動開発 – Wikipedia”. ウィキペディア (Wikipedia) :フリー百科事典. 2012-10-19. http://ja.wikipedia.org/wiki/%E3%83%81%E3%82%B1%E3%83%83%E3%83%88%E9%A7%86%E5%8B%95%E9%96%8B%E7%99%BA, (参照 2014-09-26).
9. Incept. “GitHub”. IT用語辞典 e-Words. 2014-09-28. http://e-words.jp/w/GitHub.html, (参照 2014-09-29).
10. 江口和宏, 山本竜三, 株式会社ヌーラボ. “知らないと現場で困るバージョン管理システムの基礎知識 (1/3)”. @IT. 2013-05-20. http://www.atmarkit.co.jp/ait/articles/1305/20/news015.html, (参照 2014-09-29).
11. 平屋真吾, クラスメソッド株式会社. “ガチで5分で分かる分散型バージョン管理システムGit (3/6)”. @IT. 2013-07-05. http://www.atmarkit.co.jp/ait/articles/1307/05/news028\_3.html, (参照 2014-09-29).
12. Incept. “Git”. IT用語辞典 e-Words. 2014-09-28. http://e-words.jp/w/Git.html, (参照 2014-09-29).
13. Atlassian. “Git チュートリアル”. Atlassian. https://www.atlassian.com/ja/git/tutorial, (参照 2014-10-09).
14. 大塚弘記. GitHub実践入門. 技術評論社, 2014-04-25.
15. Incept. “API”. IT用語辞典 e-Words. 2014-01-18. http://e-words.jp/w/API.html, (参照 2015-01-18).
16. GitHub. “Issues”. GitHub Developer. 2015. http://developer.GitHub.com/v3/issues/, (参照 2015-01-19).
17. adereth. “top-5000-repos.20131219.csv”. GitHub. 2013-12-23. <https://github.com/adereth/adereth.github.io/blob/master/data/top-5000-repos.20131219.csv>, (参照 2014-09-01).
18. Ubuntu Japanese Team. “Ubuntuとは”. Ubuntu Japanese Team. 2015. https://www.ubuntulinux.jp/ubuntu, (参照 2015-01-20).
19. tmizu23. “Rの自己組織化マップを利用して，確認種リストから調査地点のグループ分けを行う方法”. Hatena::Diary. 2009-12-14. <http://d.hatena.ne.jp/tmizu23/20091214/1260766698>, (参照 2014-09-29)