

目次

第 1 章 序論

1.1. 本章の構成	- 1 -
1.2. 研究背景	- 1 -
1.3. 研究目的	- 1 -
1.4. 研究方法	- 1 -
1.5. プロジェクトマネジメントとの関連性	- 2 -

第 2 章 ソフトウェア開発について

2.1. 本章の構成	- 3 -
2.2. チケットの解説	- 3 -
2.2.1. チケットの登録	- 6 -
2.2.2. チケットの一覧表示	- 7 -
2.2.3. チケットの項目	- 8 -
2.3. チケット駆動開発の解説	- 9 -
2.3.1. チケット駆動開発の開発サイクル	- 11 -
2.3.2. チケット駆動開発のメリット	- 11 -
2.3.3. チケット駆動開発のデメリット	- 11 -

第 3 章 Git について

3.1. 本章の構成	- 12 -
3.2. GitHub の解説	- 12 -
3.3. GitHub の用語	- 12 -
3.4. バージョン管理システムの解説	- 14 -
3.4.1. 集中管理方式「Subversion (SVN)」	- 14 -
3.4.2. 分散管理方式「Git」	- 15 -
3.5. Git の解説	- 16 -
3.5.1. Git コマンド	- 16 -
3.6. API (Application Programming Interface) の解説	- 32 -
3.7. GitHub API のコード	- 32 -

第 4 章 開発・調査

4.1. 本章の構成	- 34 -
4.2. 調査対象	- 34 -
4.2.1. 調査対象データ	- 34 -
4.2.2. 調査対象プロジェクト	- 36 -
4.3. 調査環境の構築	- 38 -
4.3.1. Ubuntu の解説	- 38 -
4.3.2. 調査環境の構築と処理テスト	- 39 -
4.4. 調査方法	- 42 -
4.4.1. プロジェクト成功の成否の調査	- 42 -
4.4.2. プロジェクト開始時期の調査	- 45 -
4.4.3. Issues の調査	- 46 -
4.4.4. Star 数, Contributors 数, Fork 数の調査	- 50 -
4.4.5. プロジェクトの分類	- 51 -
4.4.6. 重回帰分析	- 60 -
4.4.7. プログラムの開発	- 64 -
第 5 章 結果・考察	
5.1. 本章の構成	- 66 -
5.2. 調査結果	- 66 -
5.2.1. プロジェクト成功の成否の調査結果	- 66 -
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	- 67 -
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	- 76 -
5.3. 調査結果まとめ	- 85 -
5.4. 考察	- 86 -
参考文献	- 88 -
謝辞	- 90 -

図目次

図 1	チケットの処理フロー 参考文献 [6]	- 4 -
図 2	チケットの登録画面	- 6 -
図 3	チケットの一覧表示画面	- 7 -
図 4	集中管理方式 参考文献 [11]	- 14 -
図 5	分散管理方式 参考文献 [11]	- 15 -
図 6	git commit -amend 参考文献 [13]	- 27 -
図 7	git rebase 参考文献 [13]	- 28 -
図 8	Star 数の調査	- 42 -
図 9	リリースバージョンの調査	- 43 -
図 10	Commits 履歴の調査結果	- 45 -
図 11	線形近似曲線の式	- 48 -
図 12	Issues の調査結果	- 49 -
図 13	Star 数, Contributors 数, Fork 数の調査結果	- 50 -
図 14	RankData.csv	- 51 -
図 15	Data1 の表示	- 54 -
図 16	階層クラスター分析結果	- 55 -
図 17	非階層クラスター分析結果	- 56 -
図 18	ポジショニングマップ	- 58 -
図 19	各ユニットのオブザベーション数	- 58 -
図 20	コード情報	- 59 -
図 21	類似度の変化	- 60 -
図 22	重回帰分析結果	- 61 -
図 23	変数選択後の重回帰分析結果	- 62 -
図 24	MyModel2 の相関関係図	- 63 -
図 25	階層クラスター分析結果 上位 50 件	- 67 -
図 26	クラスターラベル 1 上位 50 件	- 68 -
図 27	クラスターラベル 2 上位 50 件	- 68 -
図 28	クラスターラベル 3 上位 50 件	- 69 -
図 29	クラスターラベル 4 上位 50 件	- 69 -
図 30	非階層クラスター分析結果 上位 50 件	- 70 -
図 31	ポジショニングマップ 上位 50 件	- 71 -
図 32	各ユニットのオブザベーション数 上位 50 件	- 72 -
図 33	コード情報 上位 50 件	- 72 -
図 34	類似度の変化 上位 50 件	- 73 -
図 35	重回帰分析結果 上位 50 件	- 74 -
図 36	相関関係図 上位 50 件	- 75 -
図 37	階層クラスター分析結果 ランダム 50 件	- 76 -
図 38	クラスターラベル 1 ランダム 50 件	- 77 -
図 39	クラスターラベル 2 ランダム 50 件	- 78 -

図 40	クラスターラベル 3 ランダム 50 件	- 78 -
図 41	クラスターラベル 4 ランダム 50 件	- 79 -
図 42	クラスターラベル 5 ランダム 50 件	- 79 -
図 43	非階層クラスター分析結果 ランダム 50 件	- 80 -
図 44	ポジショニングマップ ランダム 50 件	- 81 -
図 45	各ユニットのオブザベーション数 ランダム 50 件	- 82 -
図 46	コード情報 ランダム 50 件	- 82 -
図 47	類似度の変化 ランダム 50 件	- 83 -
図 48	重回帰分析 ランダム 50 件	- 84 -
図 49	相関関係図 ランダム 50 件	- 85 -

表目次

表 1	チケットの項目	- 8 -
表 2	GitHub の用語一覧	- 12 -
表 3	調査対象データ	- 35 -
表 4	調査対象プロジェクト 上位 50 件	- 36 -
表 5	調査対象プロジェクト ランダム 50 件	- 37 -
表 6	Star 数ランキング上位 50 件のプロジェクト (Rank.csv)	- 44 -
表 7	ランダムに選択した 50 件のプロジェクト (Random.csv)	- 44 -
表 8	Issues の累積数と経過日数	- 47 -
表 9	分析に使用する変数	- 52 -
表 10	HClust.1 のサマリ	- 55 -
表 11	HClust.1 の主成分表	- 56 -
表 12	Star 数ランキング上位 50 件のプロジェクトの成否表	- 66 -
表 13	ランダムに選択した 50 件のプロジェクトの成否表	- 66 -

第 1 章

序論

1.1. 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。

1.2. 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。

1.3. 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。

1.4. 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。

1.5. 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える。

第 2 章

チケットについて

2.1. 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．

2.2. 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．

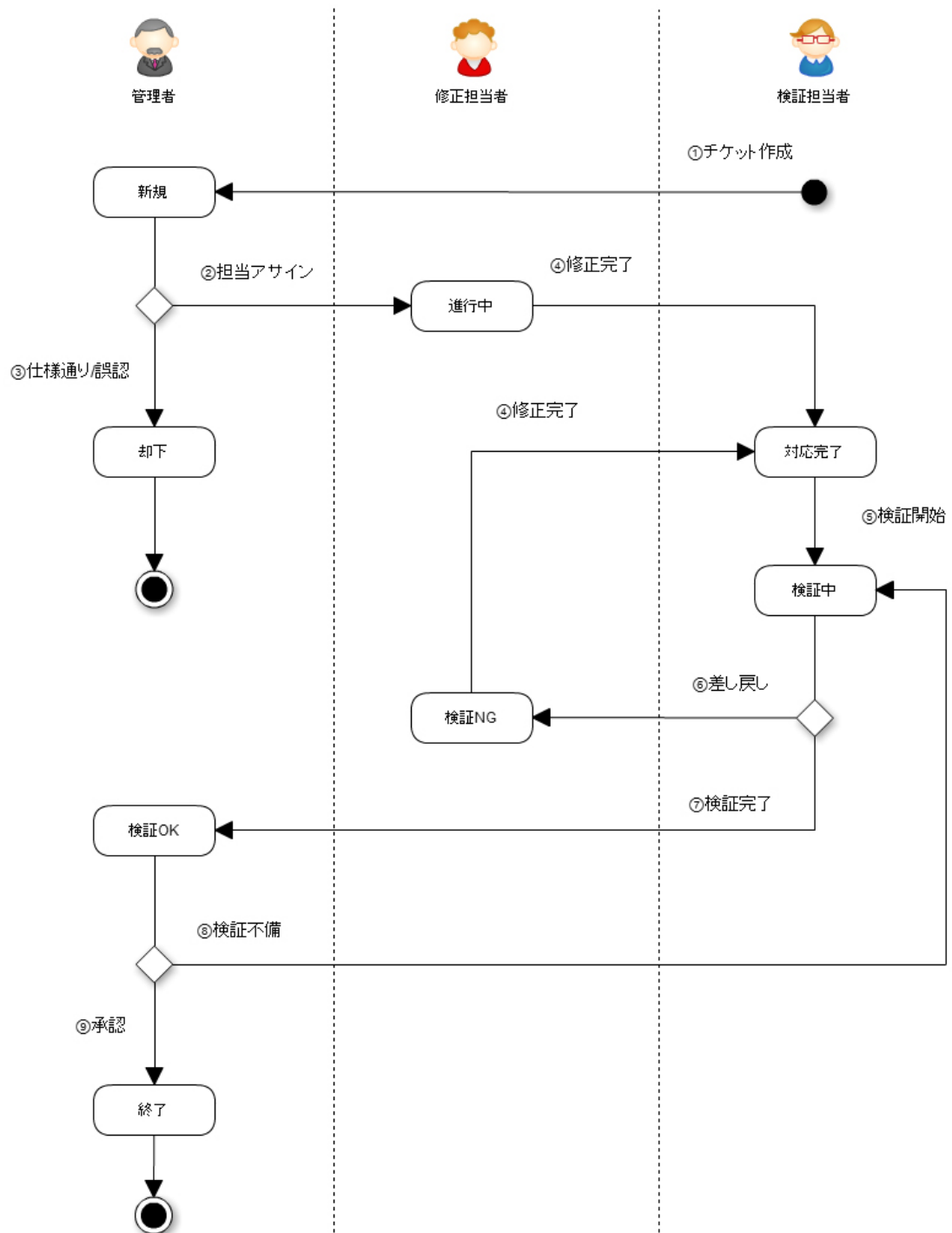


図 1 チケットの処理フロー 引用文献 [6]

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。

2) 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。

3) 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。

4) 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。

5) 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。

6) 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。

7) 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。

8) 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。

9) 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].

2.2.1. 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。

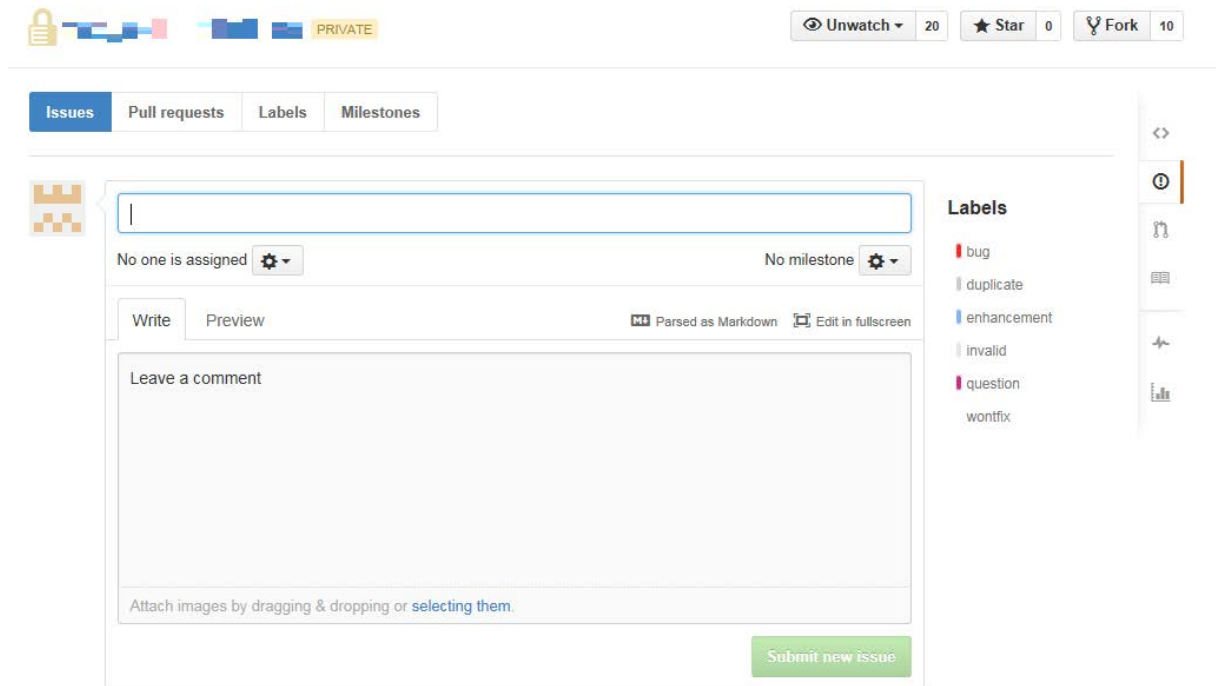


図 2 チケットの登録画面

2.2.2. 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。

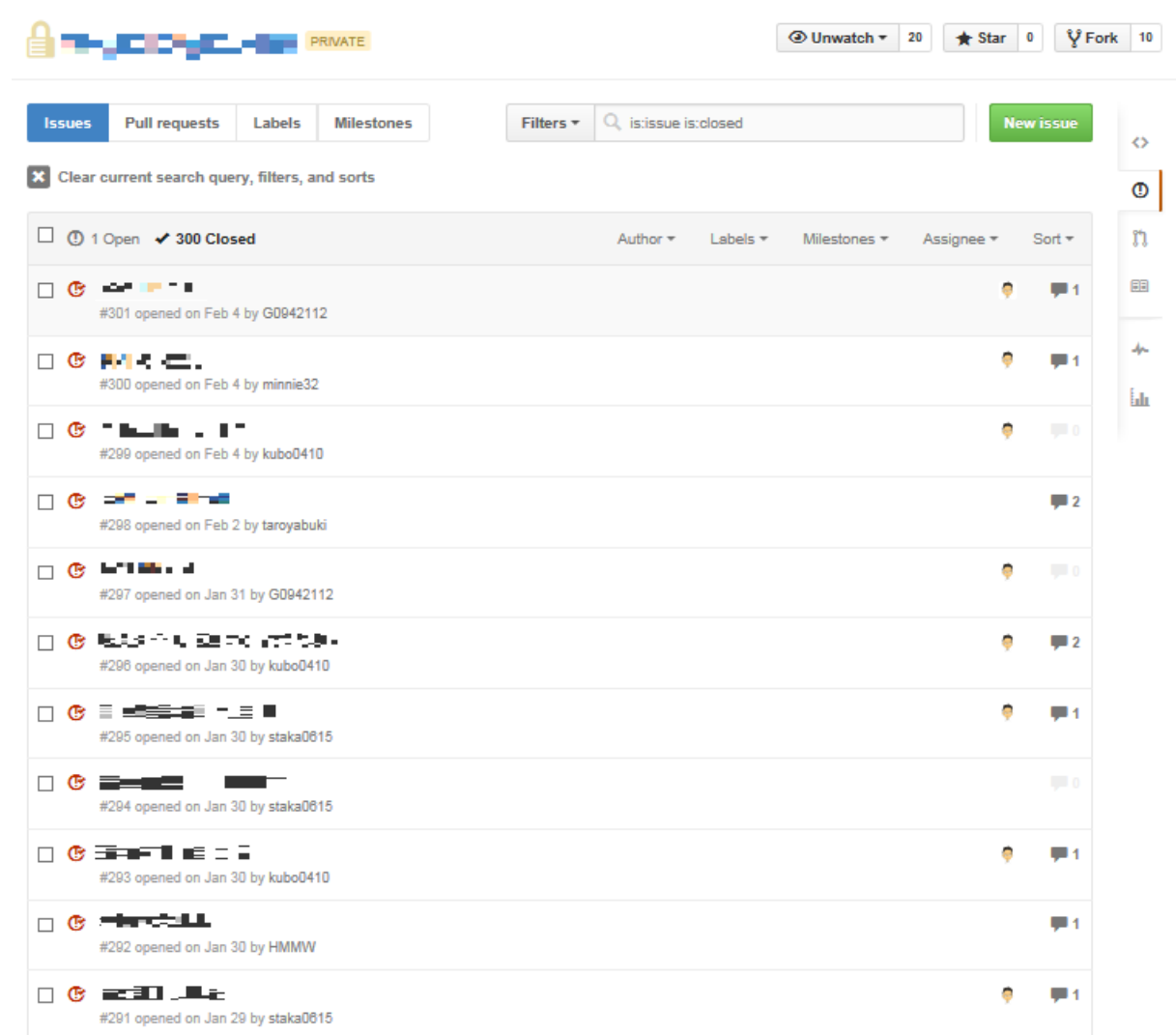


図 3 チケットの一覧表示画面

2.2.3. 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。

2.3. 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。

- 2) 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。

- 3) 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。

- 4) 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]。

2.3.1. 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]。

2.3.2. 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。

2.3.3. 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つい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

3.1. 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 についても記す。

3.2. 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]。

3.3. 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ー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。

3.4. 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]。

3.4.1. 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。

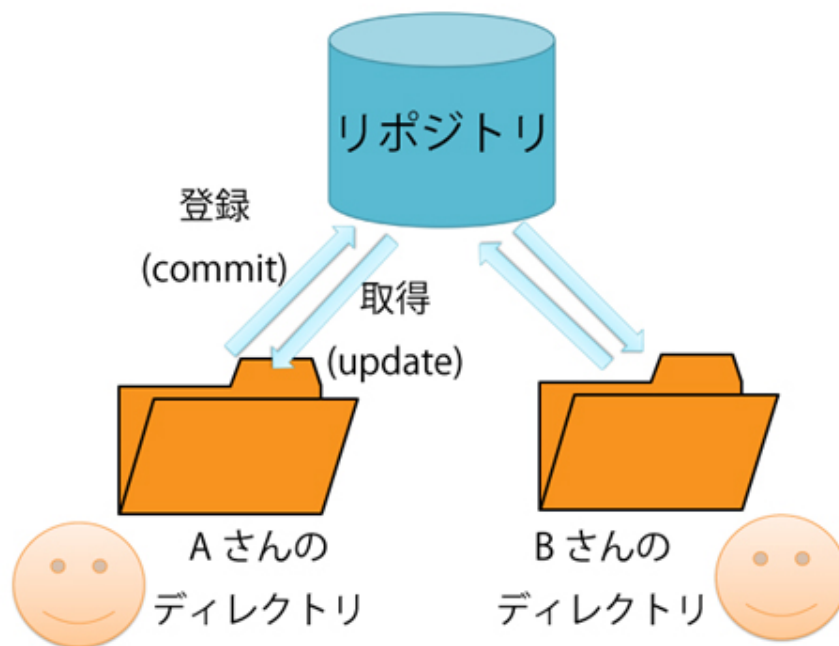


図 4 集中管理方式 引用文献 [11]

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]。

3.4.2. 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。

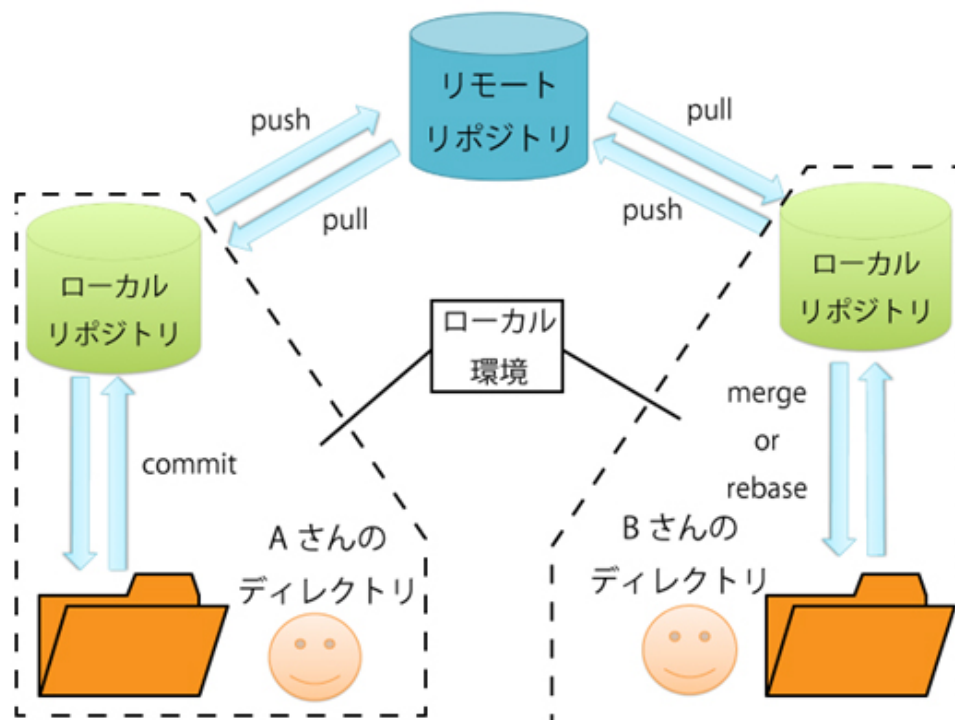


図 5 分散管理方式 引用文献 [11]

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]。

3.5. 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]。

3.5.1. 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」方法とすることができます。`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ング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ます。

```
git reset --hard
```

ステージング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象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も考えられます。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チ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られます.

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。

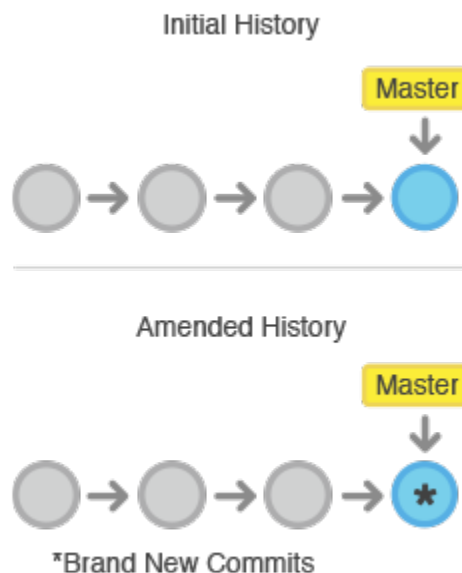


図 6 `git commit --amend` 参考文献 [13]

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
```

ステージし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無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：

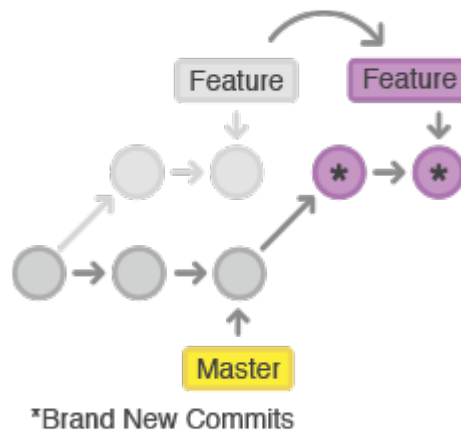


図 7 git rebase 引用文献 [13]

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様です。

```
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き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]。

3.6. 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]．

3.7. 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，本研究で使った 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

4.1. 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。

4.2. 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。

4.2.1. 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ー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。

4.2.2. 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。

4.3. 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 (ウブントウ) である．

4.3.1. Ubuntu の解説

Ubuntu (ウブントウ) 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]．

4.3.2. 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.

2) 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.

3) 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.

4) 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。

5) 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。

4.4. 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。

4.4.1. 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ければ成功とし、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。

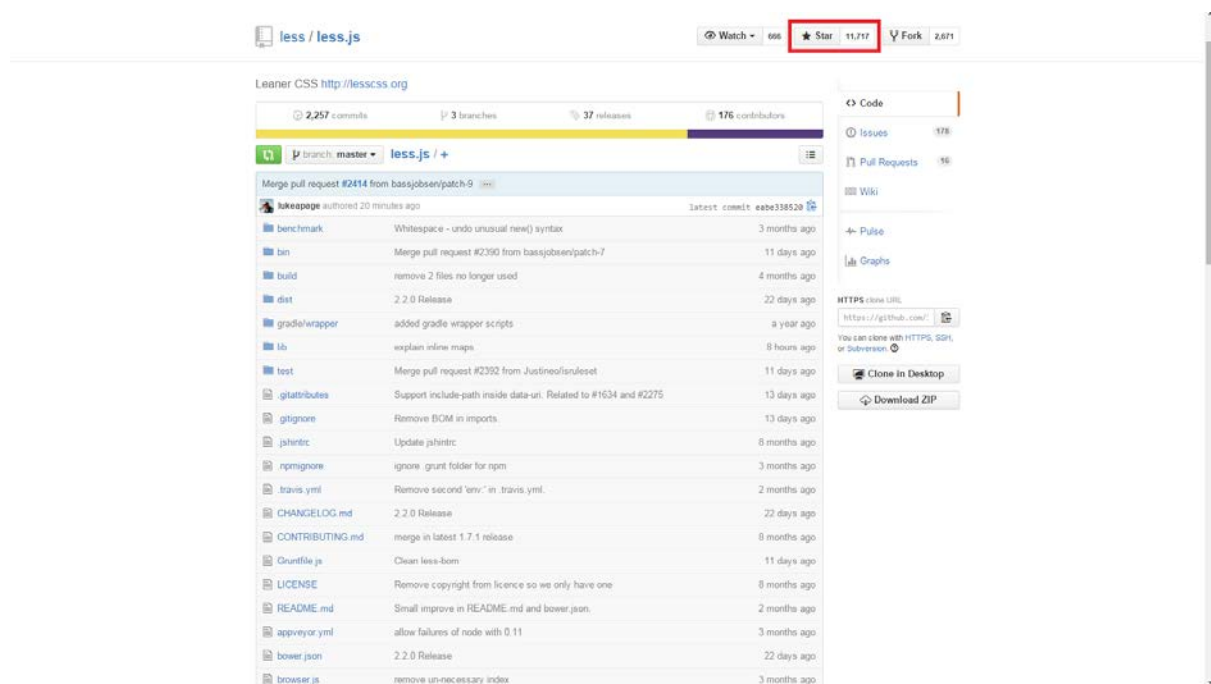


図 8 Star 数の調査

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．

2) 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．

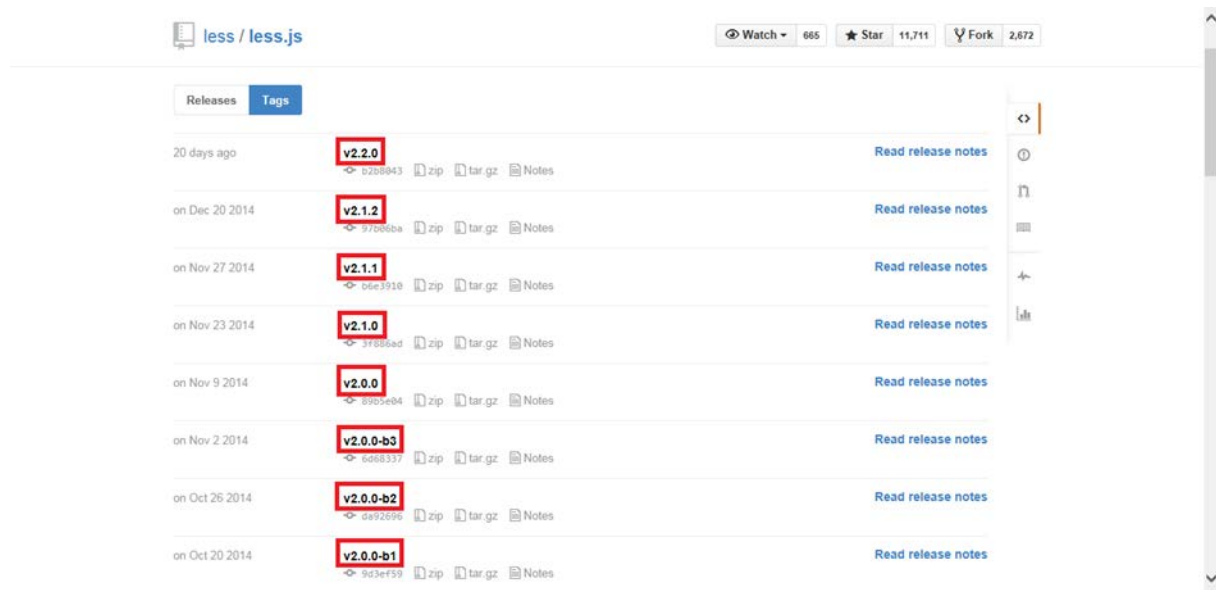


図 9 リリースバージョンの調査

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．

3) 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。

4.4.2. 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。

4.4.3. 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。

2) 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。

3) 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。

4) 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。

5) 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。

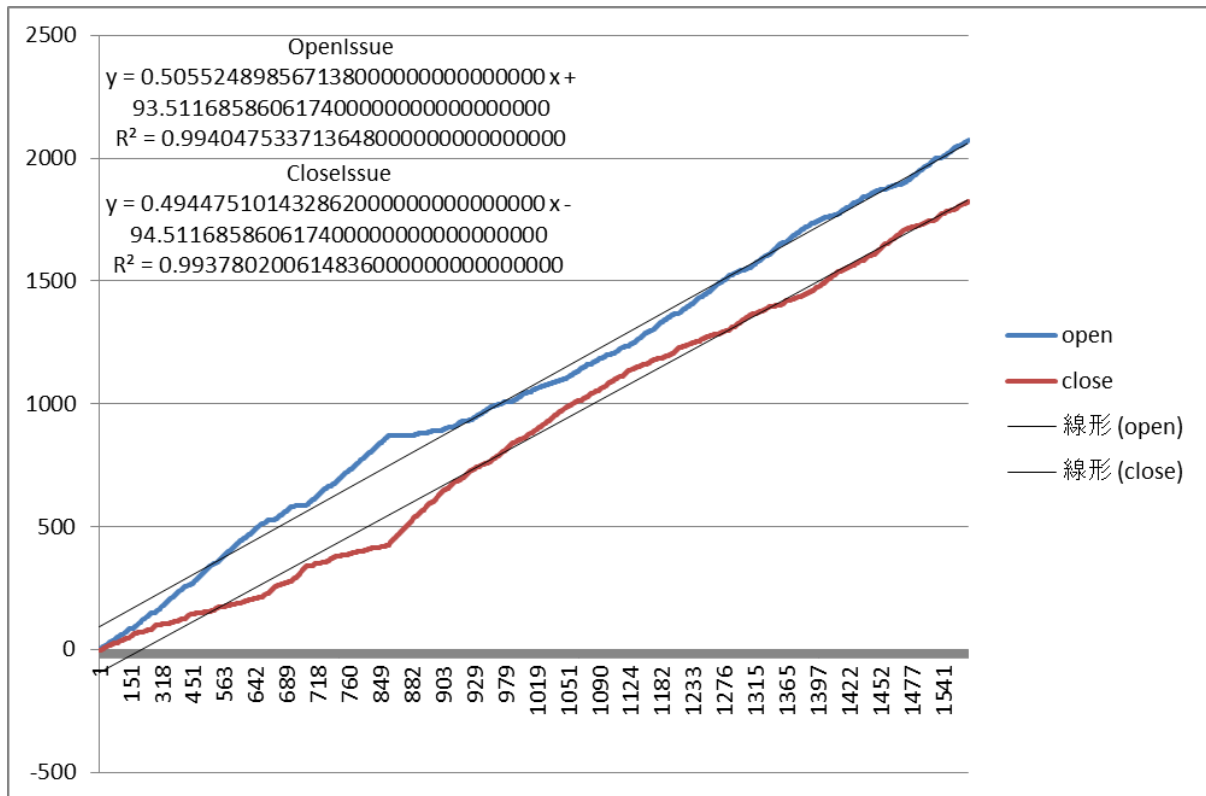


図 11 線形近似曲線の式

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。

6) “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。

ID	Created At	Closed At	Resolution Time (Days)	Title
36447000	14/06/25 03:58	14/06/25 06:22	2.400000000	Mixin guards inside mix operators?
36359050	14/06/24 07:15	14/06/24 07:27	0.200000000	two nested media queries are formulated with the screen-parameter
36300719	14/06/23 16:21	14/06/23 18:19	1.966666667	Support base64 inline sourcemaps
36234276	14/06/21 23:40	14/06/22 10:24	10.733333333	Can't specify cover or contain in background shorthand
36212781	14/06/21 00:44	14/06/22 15:15	38.516666667	Missing files in v1.7.2?
36105539	14/06/19 18:49	14/06/22 09:56	63.116666667	Don't round values returned by colour query functions.
36067933	14/06/19 10:52	14/06/21 07:27	44.583333333	Override variables with mixins
36066476	14/06/19 10:14	14/06/22 10:25	72.183333333	Rounding the result of lightness()
35968266	14/06/18 09:32	14/06/19 05:08	19.600000000	less-1.7.1.min.js not working with Lesshat mixin library 3.0
35750607	14/06/15 14:41	14/06/18 05:18	62.616666667	Numbers get rounded when using @arguments
35747444	14/06/15 11:29	14/06/15 12:11	0.700000000	Way to return sourcemap instead of inlining or writing it
35491123	14/06/11 14:48	14/06/11 16:31	1.716666667	A CSS numeric value preceded by plus sign produce weird LESS compilation error
35415692	14/06/10 19:22	14/06/10 22:59	3.416666667	Mixin duplicates selector in floatstrap (probable less issue)
35300223	14/06/09 22:13	14/06/09 23:27	1.223333333	Vereng compilation result for a fore shorthand property
35027523	14/06/05 05:02	14/06/05 11:25	6.383333333	Using @import (reference) with mixins with & selectors output incorrect cas
34941328	14/06/04 09:03	14/06/06 08:51	45.800000000	ActiveXObject in IE11: fix boolean casting
34915259	14/06/03 23:22	14/06/10 21:41	166.316666667	Base64 encode source maps
34897167	14/06/03 17:48	14/06/03 19:21	1.590000000	Can't compile .less with last Rhino (v1.7.94) and less-rhino-1.7.0.js
34748459	14/06/02 06:22	14/06/02 10:18	3.933333333	Document partials (i.e. underscore prefix on files)
34712475	14/05/31 18:40	14/06/08 16:46	190.100000000	Request to add or operator
34705799	14/05/29 12:50	14/06/02 21:56	57.100000000	ruby getting in the way of npm install
34705696	14/05/31 12:43	14/05/31 19:39	6.933333333	Scoping of mixin unlocked from namespaced mixin does not work properly
34594434	14/05/29 21:20	14/05/29 23:12	1.866666667	Remove styles assigned by class selector
34385154	14/05/27 15:44	14/05/27 17:09	1.416666667	Nested loop fails in 1.7.0
34280158	14/05/26 10:32	14/06/10 06:57	356.416666667	Sourcemap points to topmost selector of nested selectors/rules
34277526	14/05/26 01:43	14/06/06 06:53	269.166666667	Fix a bug: if the less file and line is comments, the lessc command option "mody-van" will have no effect
34179123	14/05/23 14:22	14/05/27 22:14	103.866666667	Rewriting of urls is wrong in the browser when escaping the url value
34174518	14/05/23 13:29	14/06/08 19:56	390.450000000	Travis tests are failing
34129473	14/05/22 22:09	14/05/22 22:22	0.206666667	compile error - &bar-extend(s)
34124290	14/05/22 21:32	14/05/23 13:10	15.833333333	compile error - ¬(s)
33987807	14/06/21 14:32	14/06/23 12:05	45.590000000	rhino and less don't work since 1.6.2
33986223	14/05/20 15:14	14/06/08 16:10	456.923333333	2.0.0 refactor chunker and less error
33894994	14/05/20 14:43	14/06/08 16:08	457.416666667	2.0.0 promises
33892148	14/05/20 14:15	14/06/06 07:07	400.866666667	Remove the "done!" message displayed at the end of the compilation with Rhino.
33778945	14/05/19 06:37	14/05/19 06:49	0.200000000	how to avoid color name be translated into color value?
33755773	14/05/17 17:36	14/05/20 12:00	66.400000000	fix
33685757	14/05/16 15:41	14/05/16 16:06	0.416666667	The compiler does not keep empty lines and whitespaces
33646504	14/05/16 04:16	14/05/17 16:31	36.250000000	less nested rule
33495336	14/05/14 14:10	14/05/15 06:02	15.866666667	Parametric mixing always outputs first value used
33452266	14/05/14 09:34	14/05/14 07:10	6.600000000	(Object object) output from @() on empty string
33395943	14/05/13 13:08	14/05/13 17:13	4.083333333	Passing ruleset with variables to mixin
33227368	14/05/10 03:50	14/05/10 08:01	2.183333333	Fixes #2001
33225056	14/05/10 02:23	14/05/10 06:01	3.633333333	loop within namespace fails w/ error: "Cannot call method 'concat' of undefined"
32991721	14/05/07 14:46	14/05/09 18:47	52.016666667	Destination Files on Windows only have Line Feed for Line Endings

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。

7) 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。

4.4.4. Star 数, Contributors 数, Fork 数の調査

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

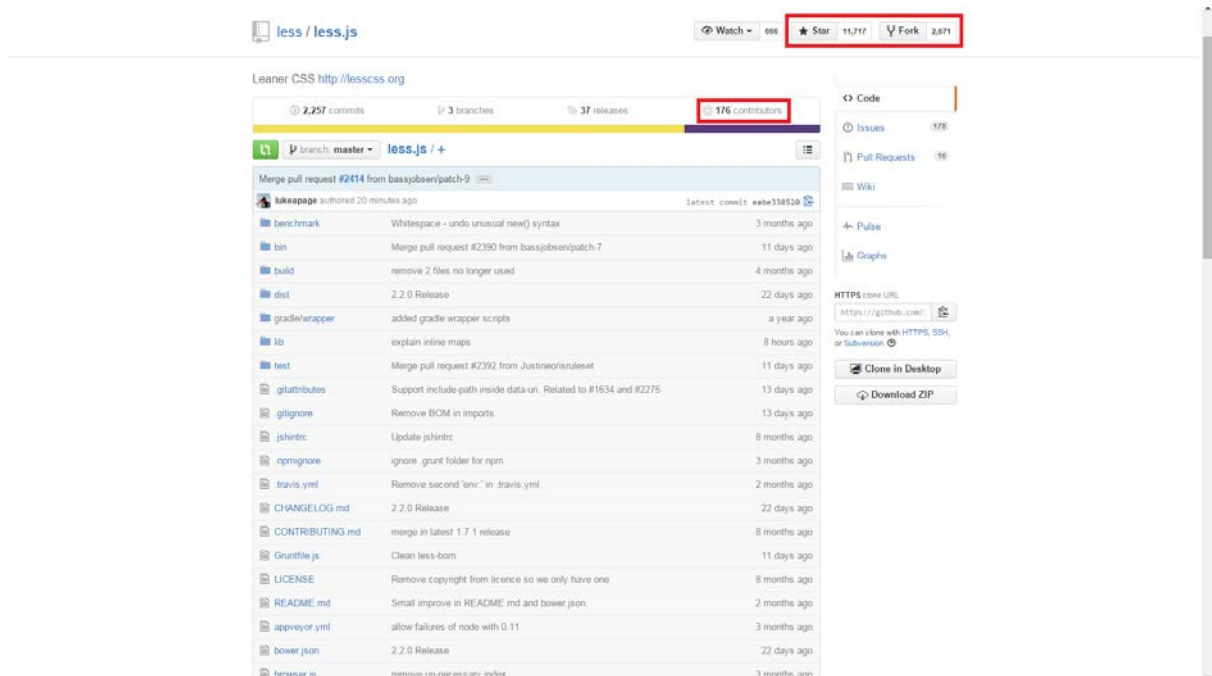
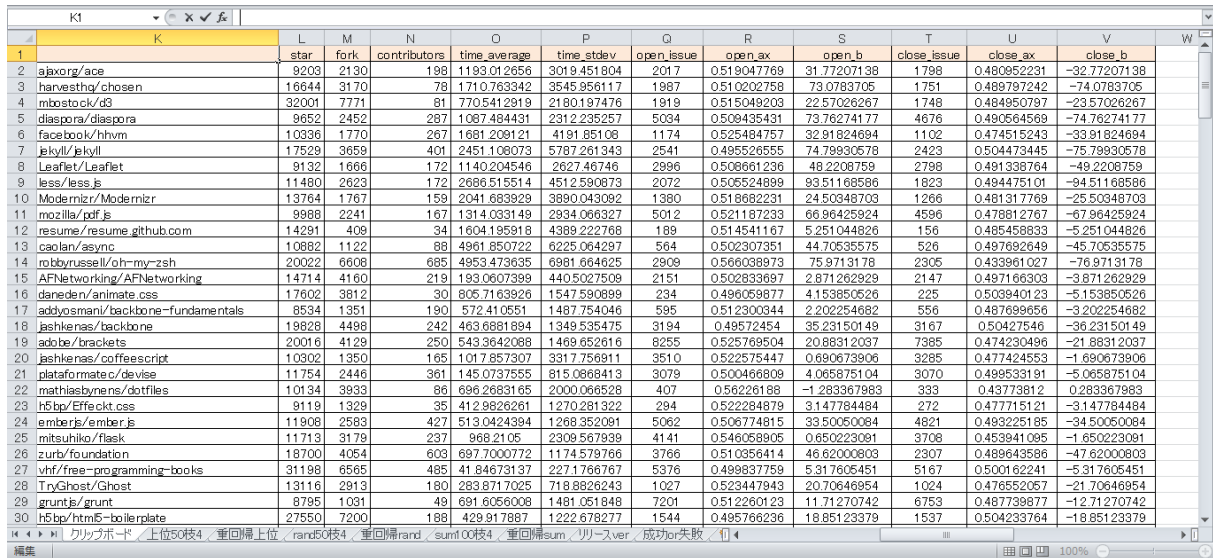


図 13 Star 数, Contributors 数, Fork 数の調査結果

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.

4.4.5. 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。



	K	L	M	N	O	P	Q	R	S	T	U	V	W
1		star	fork	contributors	time_average	time_stddev	open_issue	open_ax	open_b	close_issue	close_ax	close_b	
2	ajaxorg/ace	9203	2130	198	1193.012656	3019.451804	2017	0.519047769	31.77207138	1798	0.480952231	-32.77207138	
3	harvesthq/chosen	16644	3170	78	1710.763342	3545.956117	1987	0.510202758	73.0783705	1751	0.489797242	-74.0783705	
4	imstock/d3	32001	7771	81	770541.2919	2180.197476	1919	0.515049203	22.57026267	1748	0.484950797	-23.57026267	
5	diaspora/diaspora	9652	2452	287	1087.484431	2312.235257	5034	0.509435431	73.76274177	4676	0.490564569	-74.76274177	
6	facebook/hhvm	10396	1770	267	1681.209121	4191.85108	1174	0.525484757	32.91824694	1102	0.474515243	-33.91824694	
7	jekyll/jekyll	17529	3659	401	2451.108073	5787.261343	2541	0.495526555	74.79930578	2423	0.504473445	-75.79930578	
8	Leaflet/Leaflet	9132	1666	172	1140.204546	2627.46746	2996	0.508661236	48.2208759	2798	0.491338764	-49.2208759	
9	less/less.js	11480	2623	172	2686.515514	4512.590873	2072	0.505524899	93.51168586	1823	0.494475101	-94.51168586	
10	Modernizr/Modernizr	13764	1767	159	2041.683929	3890.043092	1380	0.518682231	24.50348703	1266	0.481317769	-25.50348703	
11	mozilla/pdf.js	9988	2241	167	1314.039149	2934.066327	5012	0.521187233	66.96425924	4596	0.478812767	-67.96425924	
12	resume/resume.github.com	14291	409	34	1604.195918	4389.222768	189	0.514541167	5.251044826	156	0.485458833	-5.251044826	
13	caolan/async	10882	1122	88	4961.850722	6225.064297	564	0.502307351	44.70535575	526	0.497692649	-45.70535575	
14	robbyrussell/oh-my-zsh	20022	6608	685	4953.473635	6981.664625	2909	0.566038973	75.9713178	2305	0.433961027	-76.9713178	
15	AFNetworking/AFNetworking	14714	4160	219	193.0607399	440.5027509	2151	0.502833697	2.871262929	2147	0.497166303	-3.871262929	
16	daneden/animate.css	17602	3812	30	805.7163926	1547.590899	234	0.496059877	4.153850526	225	0.503940123	-5.153850526	
17	addyosmani/backbone-fundamentals	8534	1351	190	572.410551	1487.754046	595	0.512300344	2.202254682	556	0.487699656	-3.202254682	
18	jashkenas/backbone	19828	4498	242	463.6881894	1349.535475	3194	0.49572454	35.23150149	3167	0.50427546	-36.23150149	
19	adobe/brackets	20016	4129	250	543.3642068	1469.652616	8255	0.525769504	20.88312037	7385	0.474230496	-21.88312037	
20	jashkenas/coffeescript	10302	1350	165	1017.857307	3317.756911	3510	0.522575447	0.690673906	3285	0.477424553	-1.690673906	
21	plataformatec/devise	11754	2446	361	145.0737555	815.0868413	3079	0.500466809	4.065875104	3070	0.499533191	-5.065875104	
22	mathiasbynens/dotfiles	10134	3933	86	696.2683165	2000.066528	407	0.56226188	-1.283367983	333	0.43773812	0.283367983	
23	h5bp/Effect.css	9119	1329	35	412.9826261	1270.281322	294	0.522284879	3.147784484	272	0.477715121	-3.147784484	
24	emberjs/ember.js	11908	2583	427	513.0424394	1268.352091	5062	0.506774815	33.50050084	4821	0.493225185	-34.50050084	
25	mitsuhiko/flask	11713	3179	237	968.2105	2309.567939	4141	0.546058905	0.650223091	3708	0.453941095	-1.650223091	
26	zurb/foundation	18700	4054	603	697.7000772	1174.579766	3766	0.510356414	46.62000803	2307	0.489643586	-47.62000803	
27	vhf/free-programming-books	31198	6565	485	41.84673137	227.1766767	5376	0.499837759	5.317605451	5167	0.500162241	-5.317605451	
28	TryGhost/Ghost	13116	2813	180	283.8717025	718.8826243	1027	0.523447943	20.70646954	1024	0.476552057	-21.70646954	
29	gruntjs/grunt	8795	1031	49	691.6056008	1481.051848	7201	0.512260123	11.71270742	6753	0.487739877	-12.71270742	
30	h5bp/html5-boilerplate	27550	7200	188	429.917887	1222.678277	1544	0.495766236	18.85123379	1537	0.504233764	-18.85123379	

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.

2) 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.

3) 変数の標準化

```
.Z <-  
scale(Data1[,c("close_ax","close_b","close_issue","open_ax","open_b","open_issue","time_aver  
age","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．

	row.names	Z.close_ax	Z.close_b	Z.close_issue	Z.open_ax	Z.open_b	Z.open_issue	Z.time_average	Z.time_stddev	var10	var11
1	ajaxorg/ace	0.1699379	0.1320647	-0.3205486	-0.1699379	-0.1447379	-0.3245614	0.03484572	0.4292287		
2	harvesthq/chosen	0.4125881	-0.7221647	-0.3299567	-0.4125881	0.7175231	-0.3304662	0.5324176	0.78235		
3	mbostock/d3	0.2796328	0.3223615	-0.3305573	-0.2796328	-0.3368239	-0.3438503	-0.3711602	-0.133651		
4	diaspora/diaspora	0.4336386	-0.7363177	0.2555485	-0.4336386	0.7318092	0.2692613	-0.06656965	-0.04509451		
5	facebook/hhvm	-0.006651664	0.1083614	-0.4598688	0.006651664	-0.1208117	-0.4904854	0.5040152	1.215545		
6	jquery/jquery	0.8152087	-0.7577542	-0.1954407	-0.8152087	0.7534473	-0.2214248	1.243908	2.285572		
7	Leaflet/Leaflet	0.4548775	-0.2081026	-0.1203759	-0.4548775	0.1986277	-0.1318692	-0.01590425	0.1663286		
8	less/less.js	0.5409185	-1.144733	-0.3155443	-0.5409185	1.144065	-0.313736	1.470141	1.430662		
9	Modernizr/Modernizr	0.1799659	0.2823817	-0.4270405	-0.1799659	-0.2964682	-0.4499393	0.8504409	1.013126		
10	mozilla/pdf.js	0.1112448	-0.5957226	0.2395347	-0.1112448	0.5898922	0.2649312	0.1511496	0.3719615		
11	resume/resume.github.com	0.2935701	0.7012096	-0.6492323	-0.2935701	-0.6983592	-0.6843586	0.4300035	1.347921		
12	caolan/async	0.6291873	-0.1354003	-0.5751683	-0.6291873	0.1252419	-0.610549	3.656797	2.579202		
13	robbyrussell/oh-my-zsh	-1.119198	-0.7819919	-0.219061	1.119198	0.7779128	-0.148993	3.648746	3.086646		
14	AFNetworking/AFNetworking	0.6147477	0.729744	-0.2506883	-0.6147477	-0.7480367	-0.2981868	-0.9261341	-1.300447		
15	daneden/animate.css	0.8005778	0.7032196	-0.6354203	-0.8005778	-0.7212629	-0.6755014	-0.3373561	-0.557934		
16	addyosmani/backbone-fundamentals	0.3550438	0.7435793	-0.5691632	-0.3550438	-0.7620021	-0.6044474	-0.561569	-0.598066		
17	jashkenas/backbone	0.8097773	0.06052243	-0.04651214	-0.8097773	-0.07252296	-0.09289773	-0.666054	-0.6907678		
18	adobe/brackets	-0.01446329	0.3572522	0.7978165	0.01446329	-0.3720426	0.9032365	-0.5894833	-0.6102065		
19	jashkenas/coffeescript	0.07316109	0.7748394	-0.02289175	-0.07316109	-0.793556	-0.03070085	-0.1334831	0.629299		
20	platformatic/devise	0.6796799	0.705039	-0.06592889	-0.6796799	-0.7230994	-0.1155327	-0.9722508	-1.049217		
21	mathiasbynens/dotfiles	-1.015579	0.8156633	-0.6138017	1.015579	-0.8347638	-0.6414506	-0.4425385	-0.254463		
22	h5bp/Effectkt.css	0.08113241	0.7447058	-0.6260122	-0.08113241	-0.7422643	-0.6636919	-0.7147834	-0.7439228		
23	emberjs/ember.js	0.5066288	0.09632016	0.2845736	-0.5066288	-0.1086573	0.2747724	-0.6186233	-0.7452167		
24	mitsuhiko/flask	-0.5710739	0.7756759	0.06178131	0.5710739	-0.7944005	0.09349608	-0.181195	-0.04688346		
25	zurk/foundation	0.4083728	-0.174996	-0.2186607	-0.4083728	0.1652099	0.01968649	-0.4411625	-0.8081089		
26	vhf/free-programming-books	0.696937	0.6998331	0.3538333	-0.696937	-0.6969697	0.3365757	-1.071455	-1.443523		
27	TryGhost/Ghost	0.04922541	0.3609054	-0.4754823	-0.04922541	-0.3757302	-0.5194187	-0.8388624	-1.113741		
28	gruntjs/grunt	0.3561472	0.5468997	0.6713073	-0.3561472	-0.5634732	0.6957823	-0.4470195	-0.6025611		
29	h5bp/html5-boilerplate	0.8086334	0.4199527	-0.3727937	-0.8086334	-0.4144579	-0.4176599	-0.6955082	-0.7758497		
30	airbnb/javascript	0.003130625	0.8145053	-0.6484316	-0.003130625	-0.8127201	-0.6867205	-0.8170194	-0.7575369		
31	defunkt/jquery-pjax	-0.3420777	0.8268158	-0.614202	0.3420777	-0.8460212	-0.6416474	0.02178665	-0.258758		
32	moment/moment	0.4344166	0.6488403	-0.3459706	-0.4344166	-0.6663723	-0.3794758	-0.641333	-0.8227996		
33	nicolas/normalize.css	0.4265014	0.7355436	-0.614202	-0.4265014	-0.7538908	-0.6538506	-0.2812519	-0.07271083		

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．

4) 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_stddev, 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．

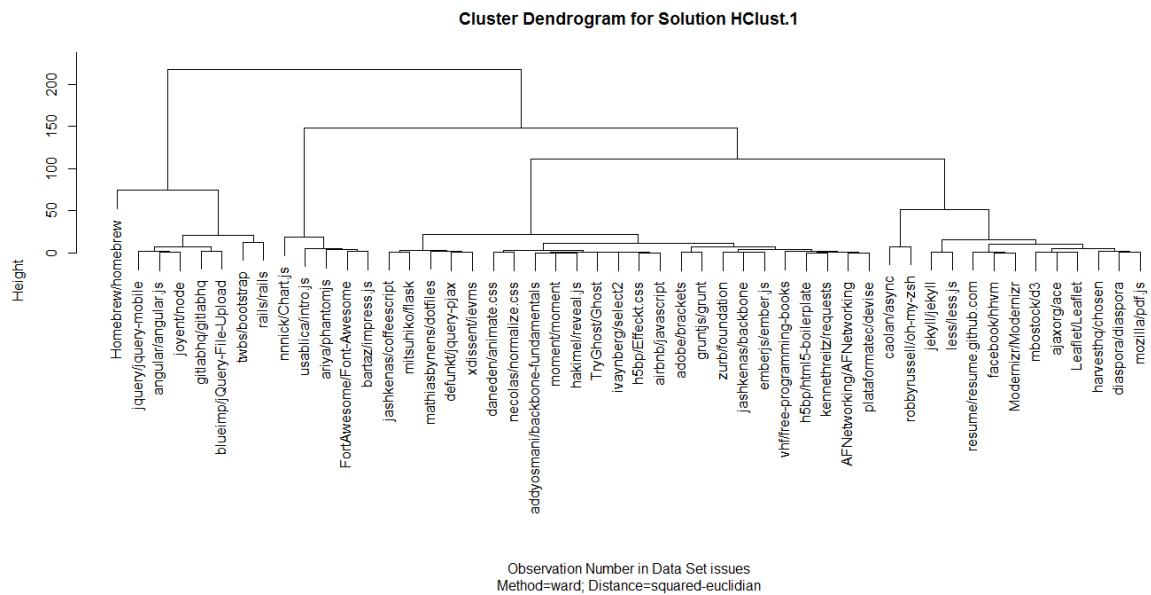


図 16 階層クラスター分析結果

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。

5) 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。

6) 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

INDICES: 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	
0.2457631	-0.2719852	-0.2652086	-0.2457631	0.2647168	-0.2704647	0.9283716	1.1176767	
INDICES: 2								
Z.close_ax	Z.close_b	Z.close_issue	Z.open_ax	Z.open_b	Z.open_issue	Z.time_average	Z.time_stdev	
0.2373109	0.5855984	-0.2340573	-0.2373109	-0.5981869	-0.2434037	-0.5272297	-0.6211431	
INDICES: 3								
Z.close_ax	Z.close_b	Z.close_issue	Z.open_ax	Z.open_b	Z.open_issue	Z.time_average	Z.time_stdev	
0.5043716	-1.7910323	1.4746005	-0.5043716	1.7964404	1.4887680	-0.2272071	-0.1523160	
INDICES: 4								
Z.close_ax	Z.close_b	Z.close_issue	Z.open_ax	Z.open_b	Z.open_issue	Z.time_average	Z.time_stdev	
-2.5850707	0.7619408	-0.5463435	2.5850707	-0.6912711	-0.5104829	0.4804677	0.3192331	

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。

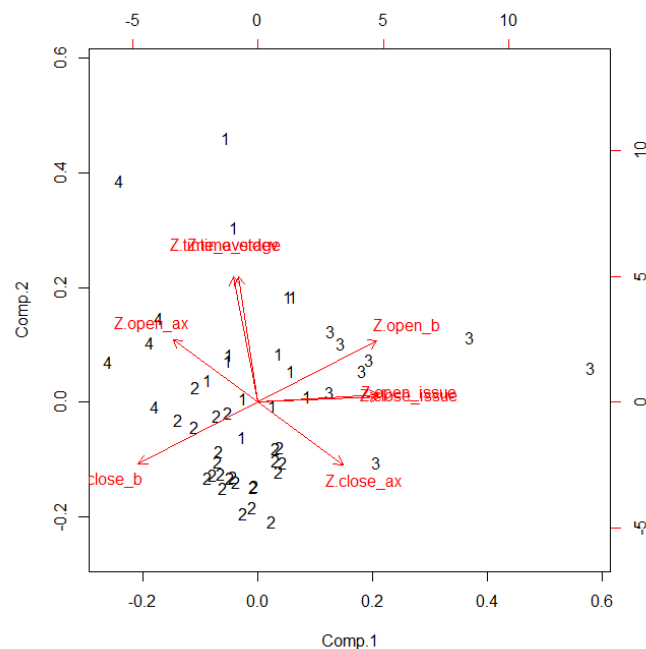


図 17 非階層クラスター分析結果

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。

7) 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。

8) 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。

9) 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。

10) 自己組織化マップの作成

```
plot(DataSOM,type="mapping",labels=row.names(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。

ポジショニングマップ

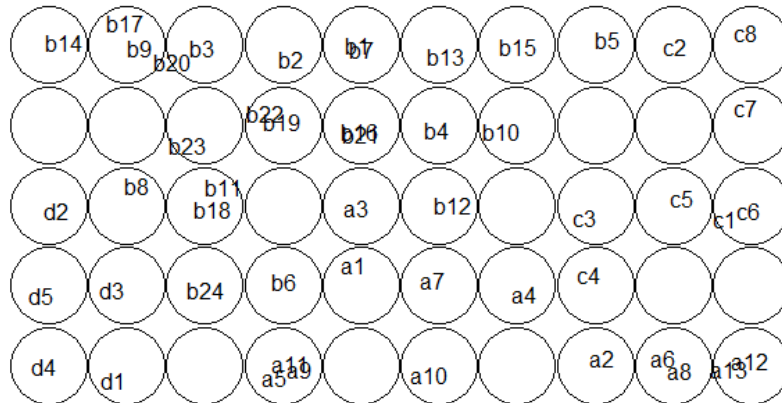


図 18 ポジショニングマップ

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．

11) 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．

各ユニットのオブザベーション数

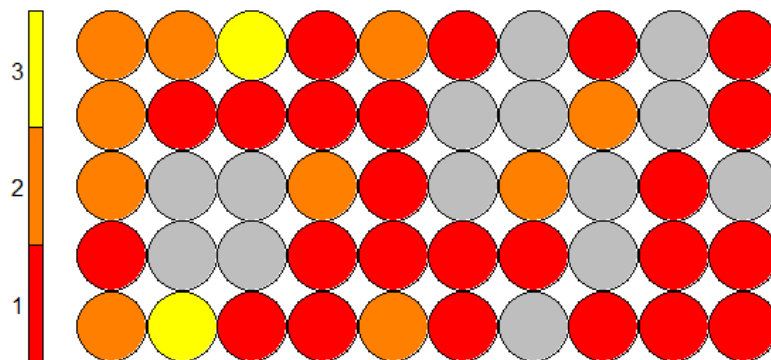


図 19 各ユニットのオブザベーション数

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．

12) 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。

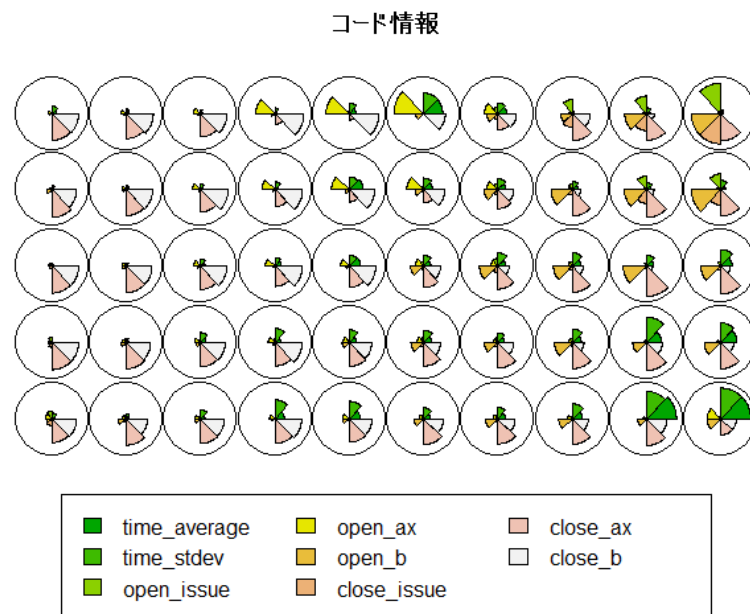


図 20 コード情報

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。

13) 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。

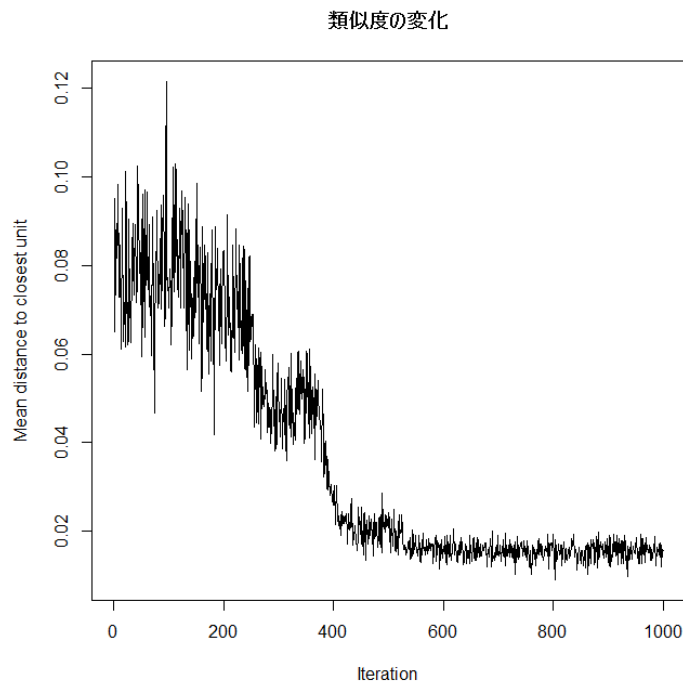


図 21 類似度の変化

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．

4.4.6. 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．

2) 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+c  
lose_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．

3) サマリの表示

```
summary(MyModel)
```

データセット“MyModel”のサマリを表示する。

「重回帰分析結果」

```
Call:
lm(formula = star ~ fork + contributors + time_average + time_stdev +
    open_issue + open_ax + open_b + close_issue + close_ax +
    close_b, data = myData)

Residuals:
    Min       1Q   Median       3Q      Max
-6956  -1938   -733   1296  12118

Coefficients: (1 not defined because of singularities)
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.664e+04  1.010e+04   1.648   0.107
fork          2.551e+00  1.575e-01  16.191 <2e-16 ***
contributors -2.015e+00  1.946e+00  -1.036   0.307
time_average -9.589e-01  1.413e+00  -0.679   0.501
time_stdev    7.595e-01  9.852e-01   0.771   0.445
open_issue    1.127e+00  1.834e+00   0.614   0.542
open_ax       -1.802e+04  1.952e+04  -0.923   0.361
open_b         1.998e+02  2.039e+02   0.980   0.333
close_issue   -1.187e+00  1.875e+00  -0.633   0.531
close_ax              NA              NA      NA      NA
close_b        2.178e+02  2.055e+02   1.059   0.296
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3758 on 40 degrees of freedom
Multiple R-squared:  0.9027,    Adjusted R-squared:  0.8808
F-statistic: 41.25 on 9 and 40 DF,  p-value: < 2.2e-16
```

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。

4) 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。

```

Call:
lm(formula = star ~ fork + contributors, data = myData)

Residuals:
    Min       1Q   Median       3Q      Max
-7161.8 -2237.6  -888.9   1487.8 10649.7

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  7621.181    706.6465   10.785 2.64e-14 ***
fork           2.5245      0.1295   19.494 < 2e-16 ***
contributors  -2.9547      0.8133   -3.633 0.000692 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3596 on 47 degrees of freedom
Multiple R-squared:  0.8953,    Adjusted R-squared:  0.8909
F-statistic: 201 on 2 and 47 DF,  p-value: < 2.2e-16

```

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 $y = 2.5245x_1 - 2.9547x_2 + 7621.1818$ である。Star 数を求めるには、 x_1 に Fork 数を代入し、 x_2 に Contributors 数を代入する。

5) 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。

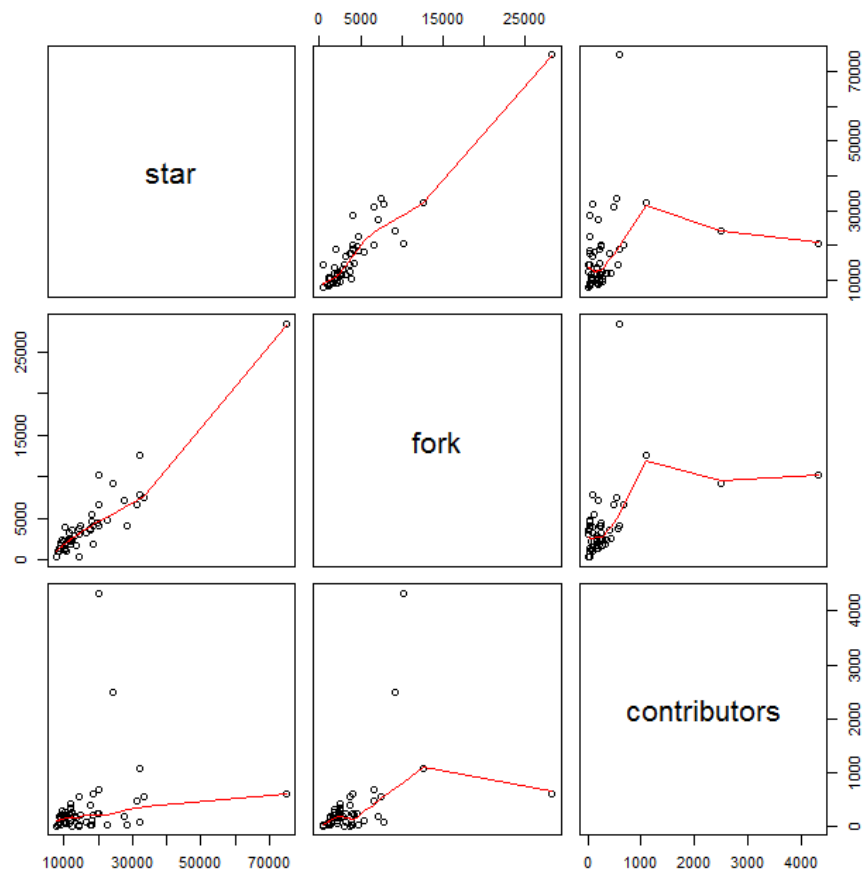


図 24 MyModel2 の相関関係図

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．

4.4.7. 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 of 省略, および各種設定を行う.

2) 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

5.1. 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。

5.2. 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。

5.2.1. 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。

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

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。

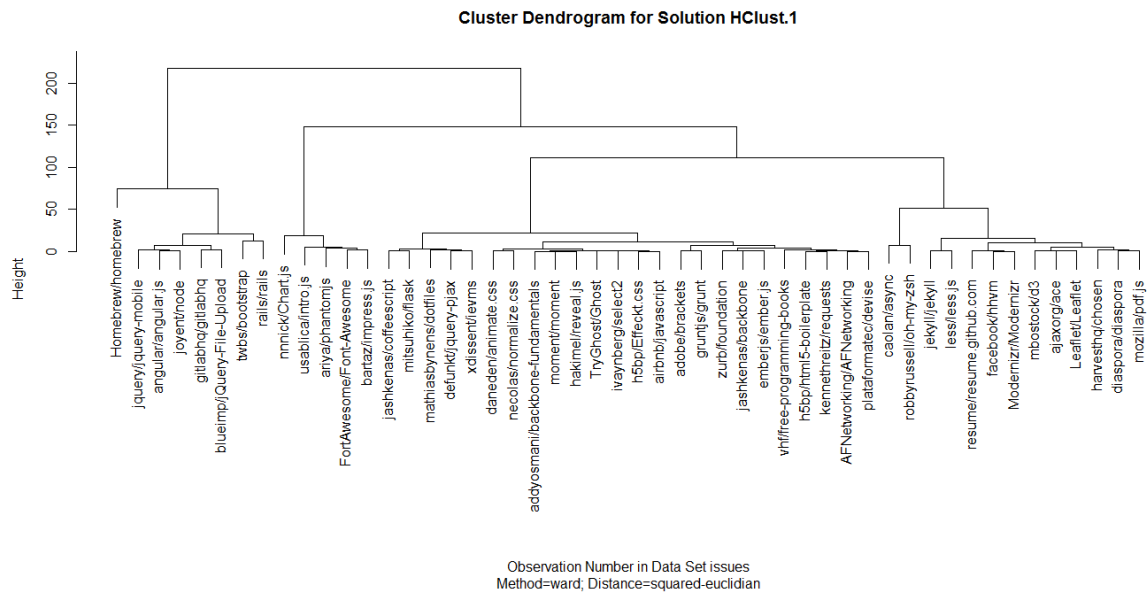


図 25 階層クラスター分析結果 上位 50 件

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。

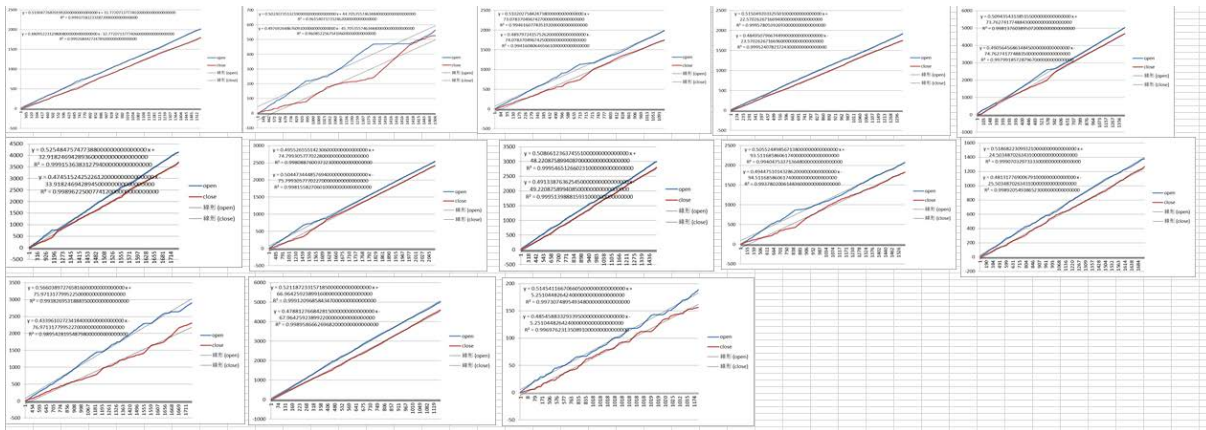


図 26 クラスタラベル 1 上位 50 件

図 26 はクラスタラベル 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.

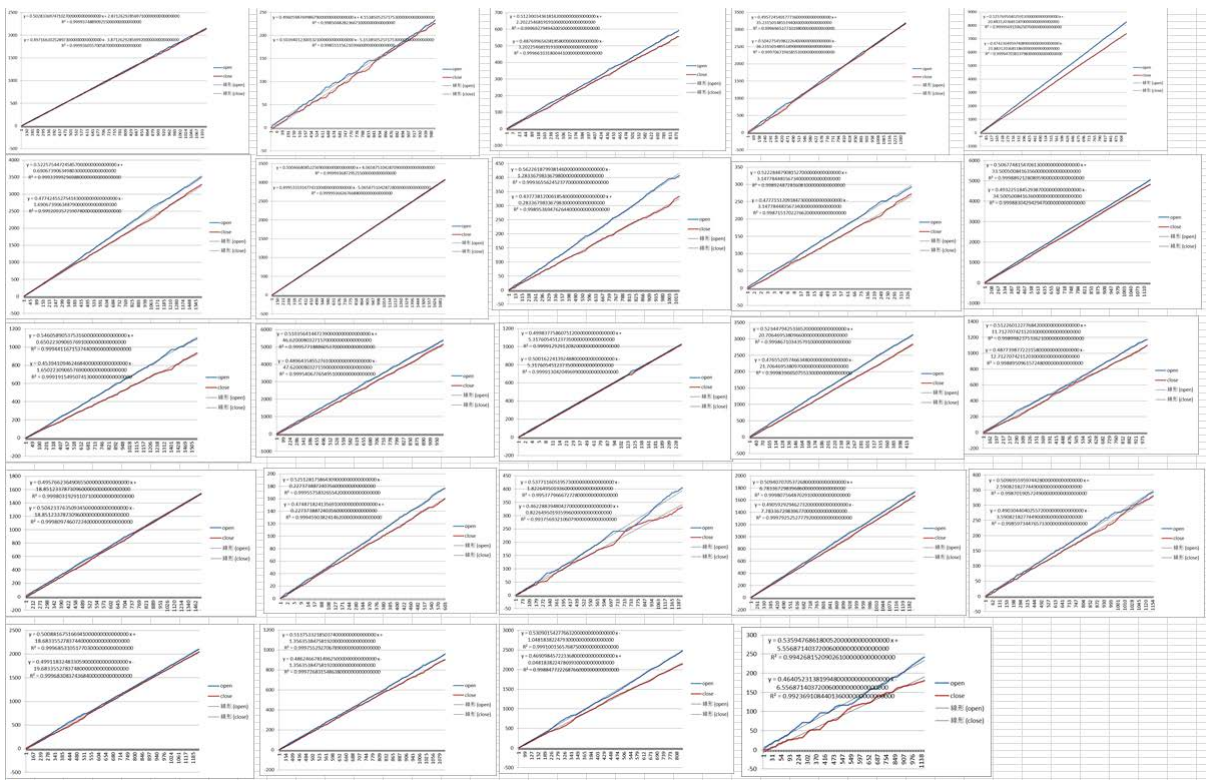


図 27 クラスタラベル 2 上位 50 件

図 27 はクラスタラベル 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.

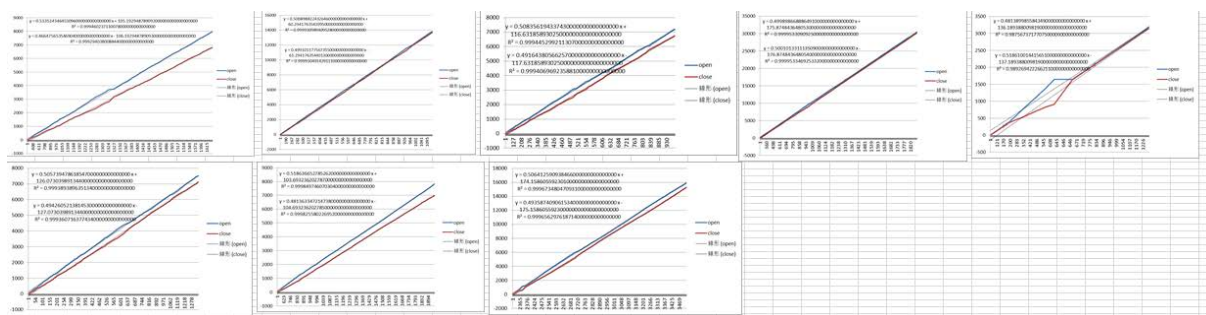


図 28 クラスターラベル 3 上位 50 件

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．

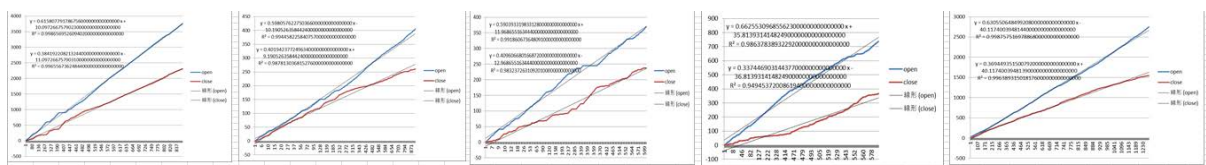


図 29 クラスターラベル 4 上位 50 件

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．

2) 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.

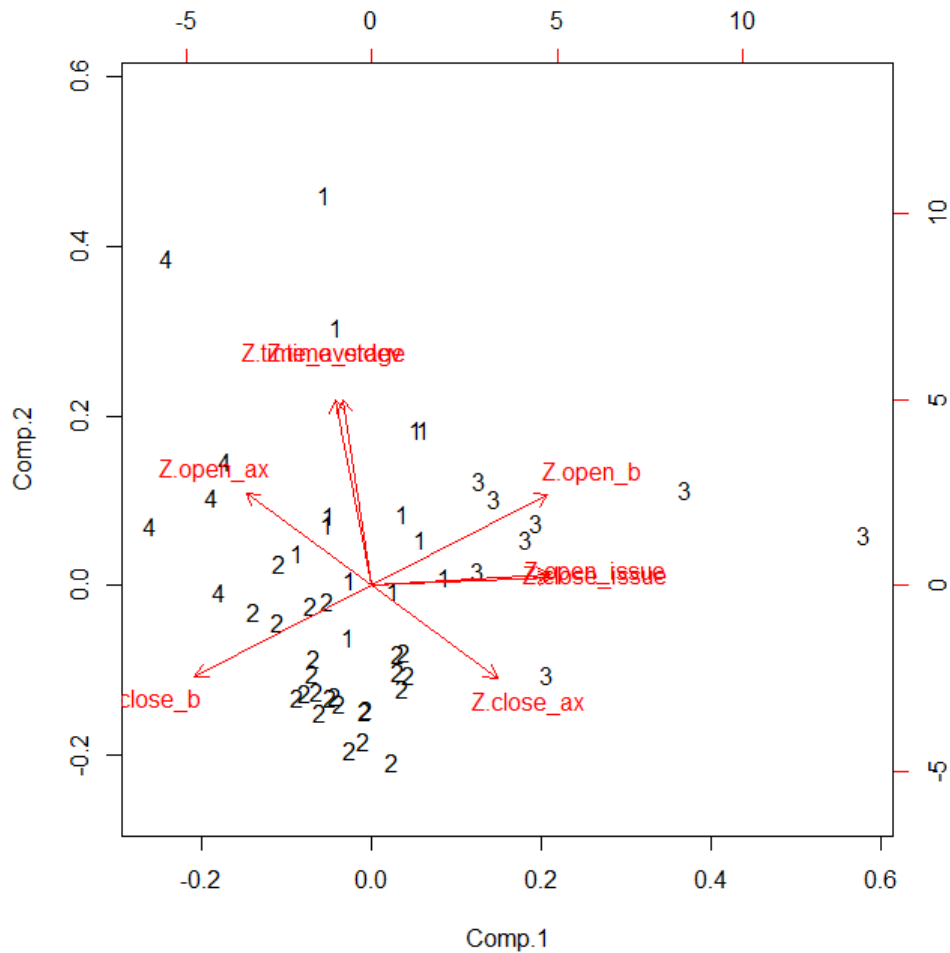


図 30 非階層クラスター分析結果 上位 50 件

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。

3) 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．

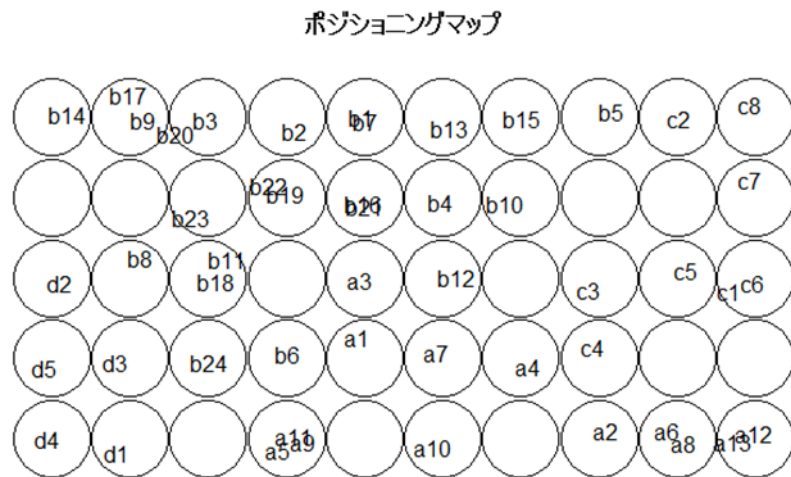


図 31 ポジショニングマップ 上位 50 件

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．

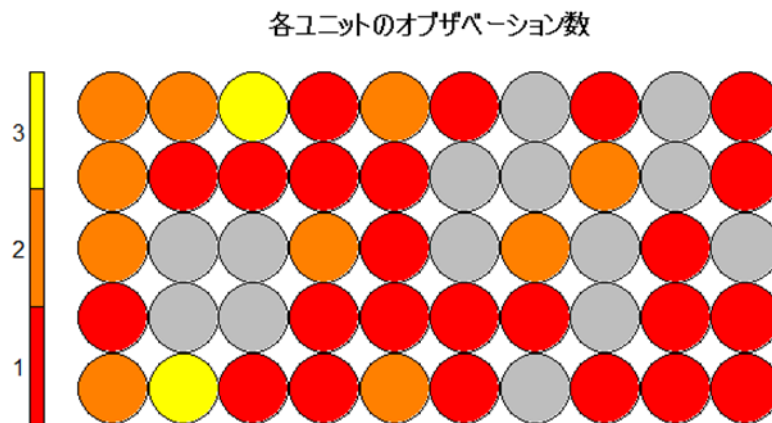


図 32 各ユニットのオブザベーション数 上位 50 件

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。

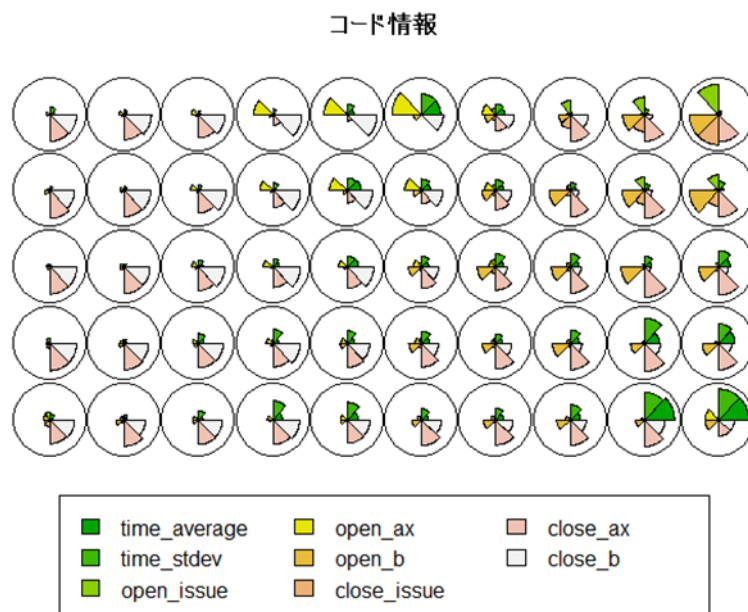


図 33 コード情報 上位 50 件

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。

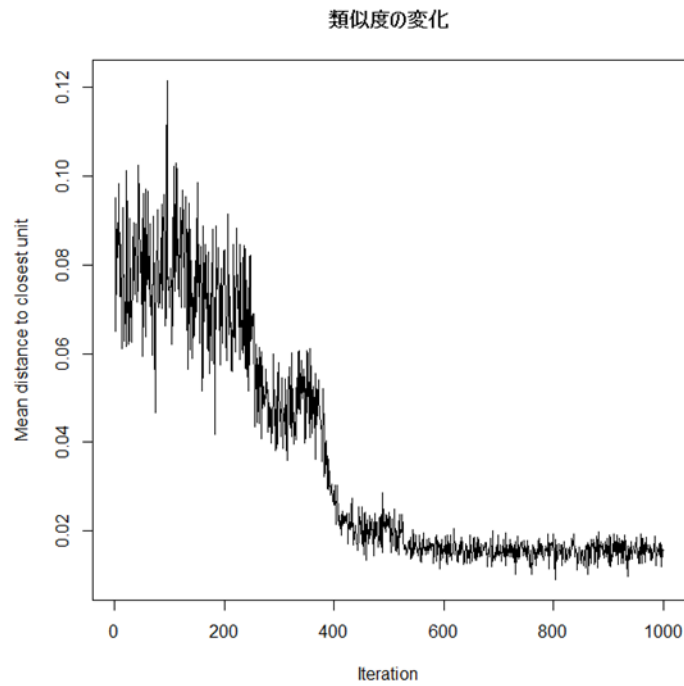


図 34 類似度の変化 上位 50 件

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．

4) 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。

```
Call:
lm(formula = star ~ fork + contributors, data = myData)

Residuals:
    Min       1Q   Median       3Q      Max
-7161.8 -2237.6  -888.9   1487.8 10649.7

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  7621.181    706.6465   10.785 2.64e-14 ***
fork           2.5245     0.1295   19.494 < 2e-16 ***
contributors  -2.9547     0.8133   -3.633 0.000692 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3596 on 47 degrees of freedom
Multiple R-squared:  0.8953,    Adjusted R-squared:  0.8909
F-statistic: 201 on 2 and 47 DF,  p-value: < 2.2e-16
```

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 $y = 2.5245x_1 - 2.9547x_2 + 7621.1818$ であり、Fork 数と Contributors 数は逆相関の関係にある。

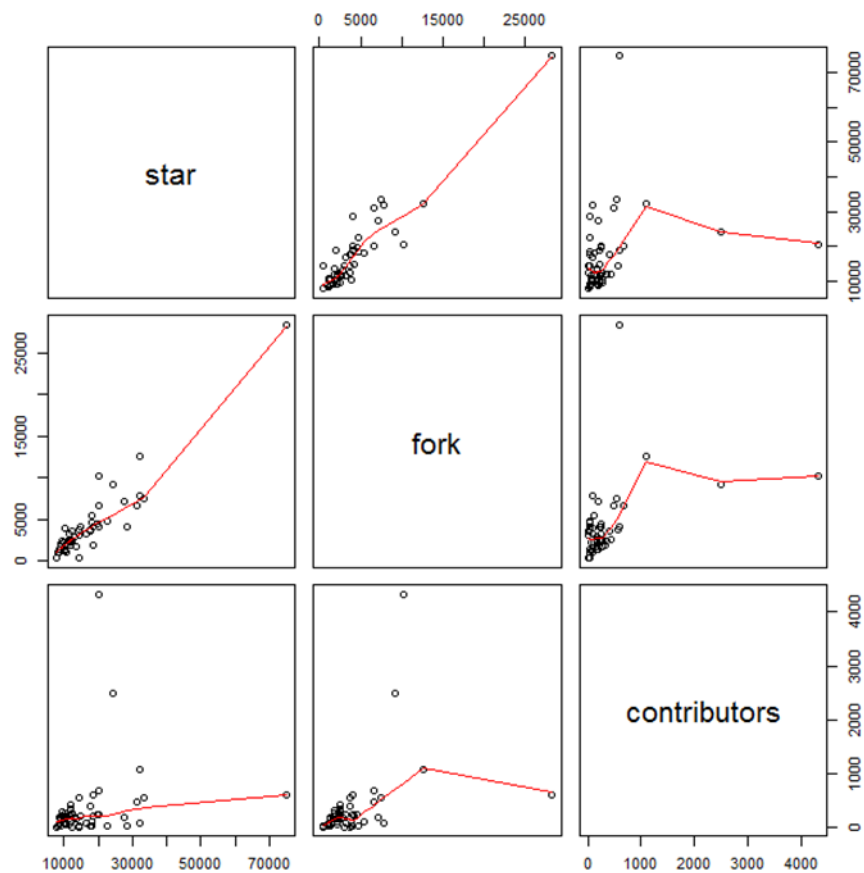


図 36 相関関係図 上位 50 件

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．

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

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。

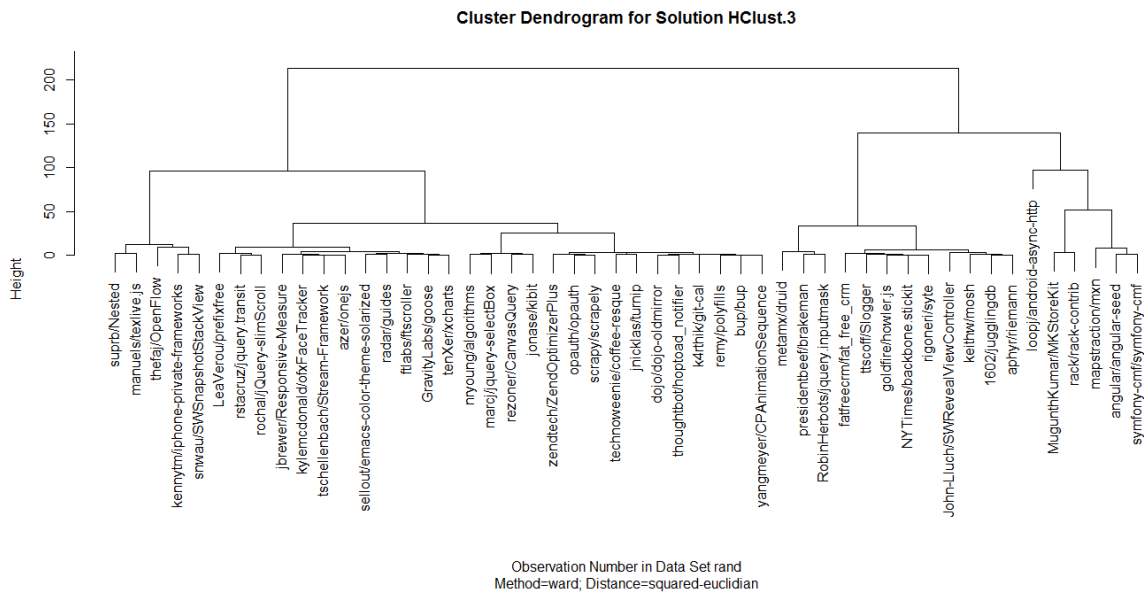


図 37 階層クラスター分析結果 ランダム 50 件

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。

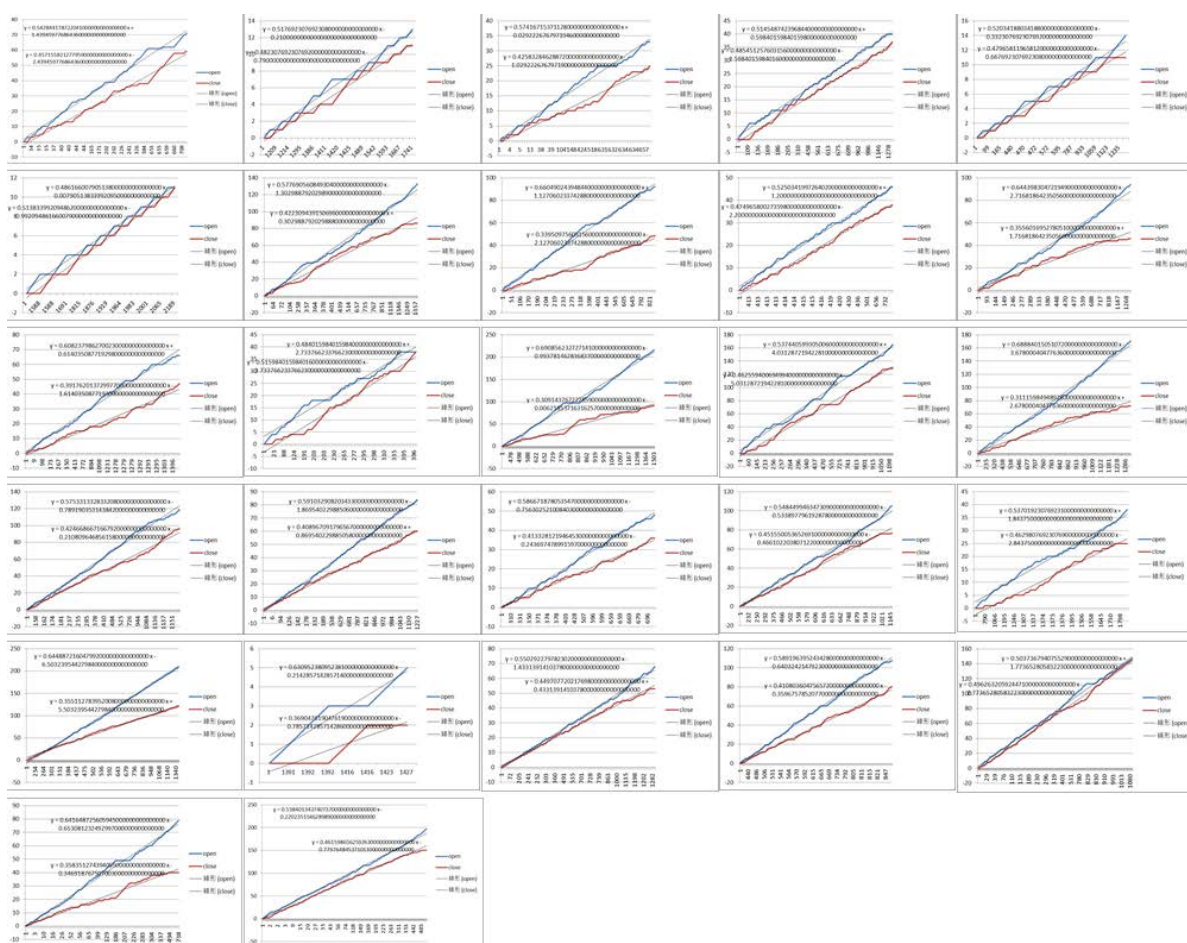


図 38 クラスタラベル 1 ランダム 50 件

図 38 はクラスタラベル 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．

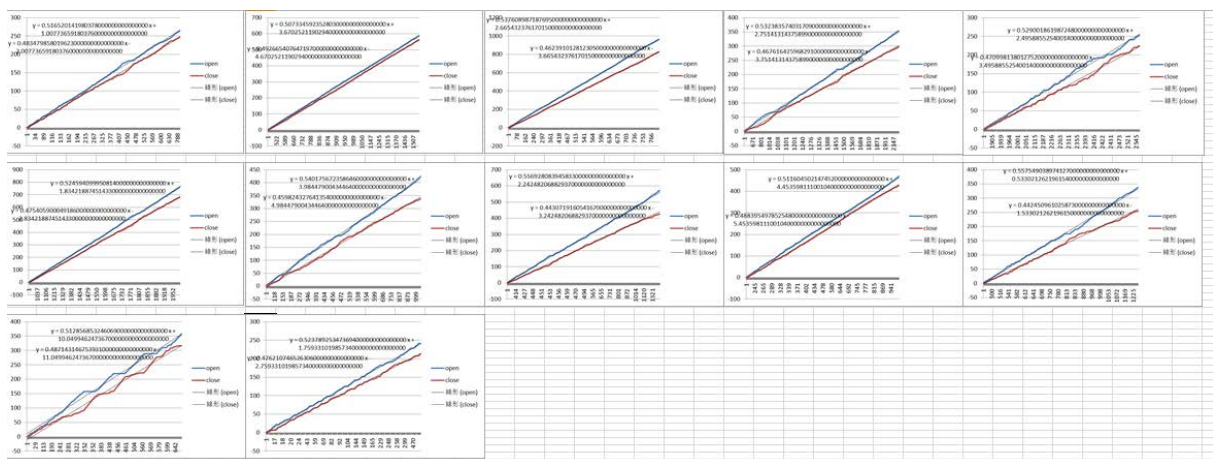


図 41 クラスターラベル 4 ランダム 50 件

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．

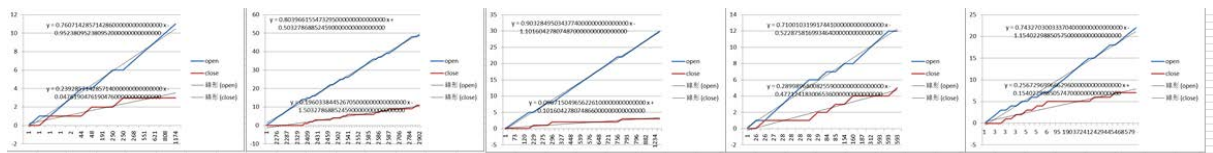


図 42 クラスターラベル 5 ランダム 50 件

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．

2) 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．

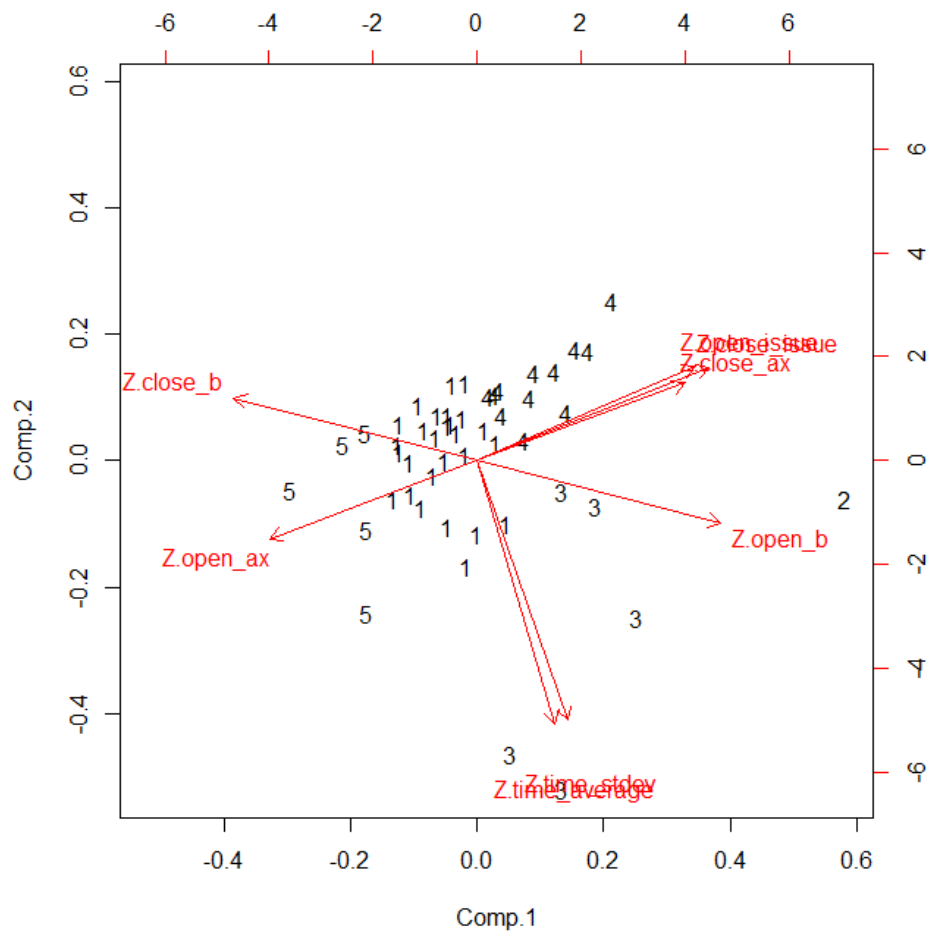


図 43 非階層クラスター分析結果 ランダム 50 件

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．

3) 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．

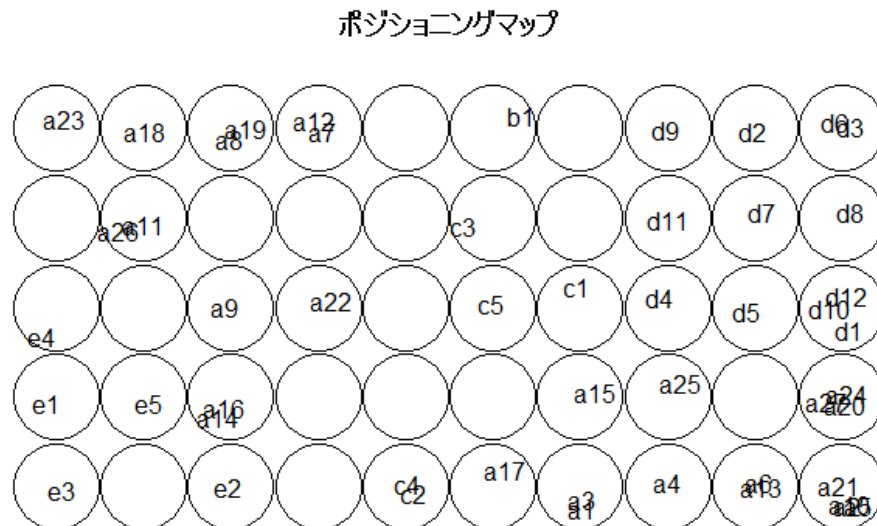


図 44 ポジショニングマップ ランダム 50 件

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．

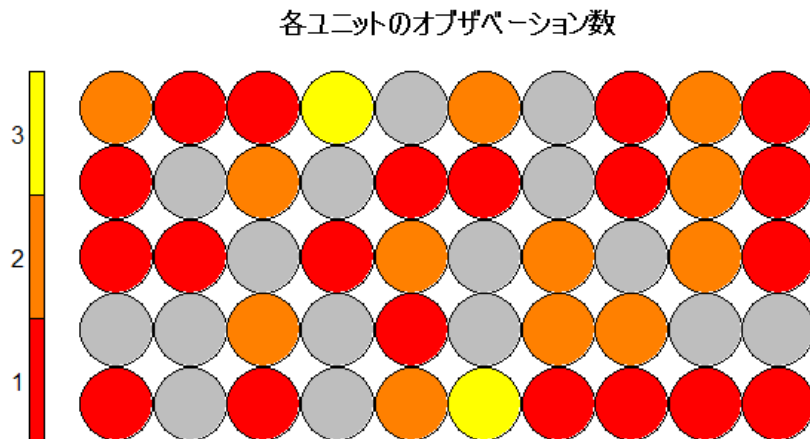


図 45 各ユニットのオブザベーション数 ランダム 50 件

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。

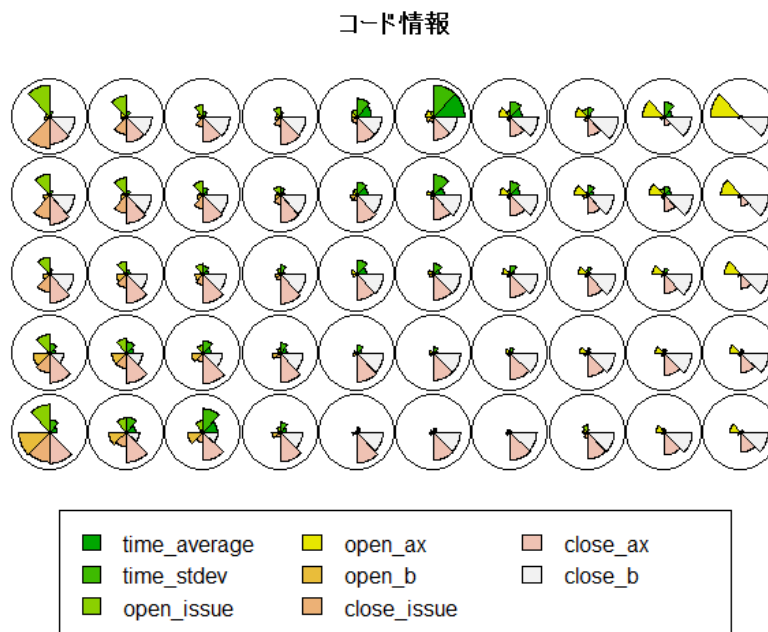


図 46 コード情報 ランダム 50 件

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。

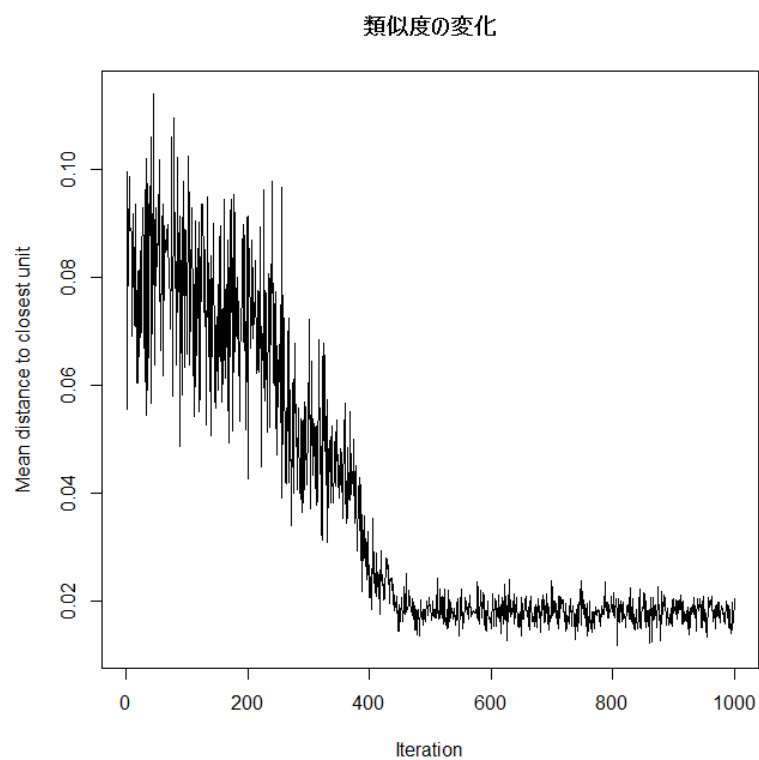


図 47 類似度の変化 ランダム 50 件

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．

4) 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.

```
Call:
lm(formula = star ~ fork + contributors + open_issue + close_issue,
    data = myData)

Residuals:
    Min       1Q   Median       3Q      Max
-1378.7  -458.8  -106.1   226.1  2646.5

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  403.1287    192.5519   2.094  0.04196 *
fork          1.6254     0.1756   9.257 5.52e-12 ***
contributors  9.3808     6.6872   1.403  0.16754
open_issue   11.3685     3.6405   3.123  0.00313 **
close_issue  -12.2534     4.0558  -3.021  0.00414 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 772.5 on 45 degrees of freedom
Multiple R-squared:  0.727,    Adjusted R-squared:  0.7027
F-statistic: 29.96 on 4 and 45 DF,  p-value: 3.579e-12
```

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 $y = 1.6254x_1 + 9.3808x_2 + 11.3685x_3 - 12.2534x_4 + 403.1287$ であり、ClosedIssues 数は Fork 数、Contributors 数、OpenIssues 数と逆相関の関係にある。

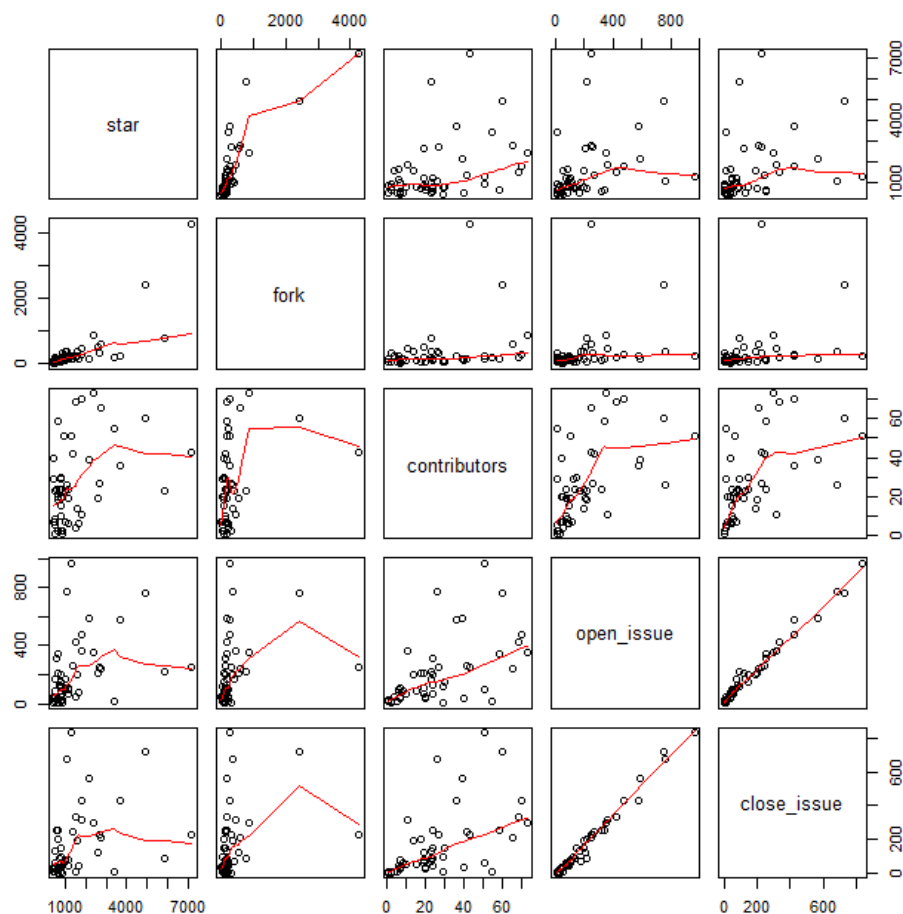


図 49 相関関係図 ランダム 50 件

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．

5.3. 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．

5.4. 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。

2) 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。

3) 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. “知らないで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.ubuntu-linux.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)

謝辞

本研究を進めるに当たり，ご指導を頂いた卒業論文指導教員の矢吹太郎準教授に感謝致します．また，日常の議論を通じて多くの知識や示唆を頂いた皆様に感謝します．