**Magistrale in Ingegneria Informatica**



STABILITà del progetto E Predizione di bug di classi java in progetti jira

ISW2 2019-2020

Valentino Perrone 0264636

Indice

[Capitolo1 Introduzione 2](#_Toc44015431)

[1.1 Tecnologie usate 2](#_Toc44015432)

[1.2 Organizzazione codice ad alto livello 2](#_Toc44015433)

[Capitolo2 Delivery1 3](#_Toc44015434)

[2.1 Descrizione del problema 3](#_Toc44015435)

[2.2 Analisi della stabilità 3](#_Toc44015436)

[2.3 Ulteriori analisi 6](#_Toc44015437)

[2.3.1 Parquet-mr default branch 7](#_Toc44015438)

[2.3.2 Parquet-mr tutti i branch 8](#_Toc44015439)

[2.3.3 Tutti repo default branch 9](#_Toc44015440)

[2.3.4 Tutti repo tutti i branch 11](#_Toc44015441)

[2.3.4 Considerazioni 13](#_Toc44015442)

[2.4 Dettagli implementativi 13](#_Toc44015443)

[2.5 Conclusioni 14](#_Toc44015444)

[Capitolo3 Delivery2 15](#_Toc44015445)

[3.1 Descrizione del problema 15](#_Toc44015446)

[3.2 Costruzione del dataset 15](#_Toc44015447)

[3.2.1 Dettagli sul calcolo delle metriche 16](#_Toc44015448)

[3.2.2 Calcolo del LifeCycle 16](#_Toc44015449)

[3.2.3 Bug da escludere 17](#_Toc44015450)

[3.2.4 Tecniche per calcolare P 17](#_Toc44015451)

[3.2.5 Classi buggy 17](#_Toc44015452)

[3.3 Dettagli implementativi 17](#_Toc44015453)

[Riferimenti sul Web 18](#_Toc44015454)

# Capitolo1 Introduzione

Gli obiettivi dello studio descritto in questo documento rappresentano la stabilità del progetto considerato attraverso la tecnica dello statistical control process e predizione di bug in classi java in progetti gestiti dal sistema di ticketing Jira e di versioning Git.

I progetti esaminati sono i seguenti:

1. Apache Bookkeeper
2. Apache Parquet
3. Apache Syncope

Tale studio si compone di due parti:

* Delivery1: riguarda la stabilità del progetto preso in considerazione
* Delivery2: si focalizza sul confronto di tecniche di machine learning applicate alla predizione dei bug di classi Java

## Tecnologie usate

Il software utilizzato per lo studio è stato scritto in linguaggio Java8 su IDE Eclipse. E’ stato utilizzato SonarCloud come piattaforma per il rilevamento dei difetti relativi alla qualità del software, Travis CI come builder di progetto ed infine SVN connesso alla piattaforma GITHUB come sistema di versioning. Per quanto riguarda la visualizzazione dei risultati si è scelto di utilizzare Excel.

## Organizzazione codice ad alto livello

Il codice scritto in java è stato organizzato in 3 package principali:

1. common: codice comune ad entrambe le delivery
   1. entity
   2. gestione directory per scaricare il repository
   3. parser git e jira
   4. parser output da rest api
   5. classi utils
   6. stringhe
2. firstdelivery: codice relativo alla prima consegna
   1. main
   2. controller
   3. classe output
   4. gestore scritture su file
   5. classi utils
3. secondelivery: codice relativo alla seconda consegna
   1. main
   2. controller
   3. calcolatore life cycle
   4. calcolatore proportion method
   5. calcolatore metriche
   6. gestore scritture su file
   7. classi machine learning
   8. stringhe

# Capitolo2 Delivery1

## 2.1 Descrizione del problema

La prima consegna richiesta, come detto precedentemente riguarda la stabilità del progetto. Un processo è statisticamente stabile se il numero di bug è sempre limitato superiormente ed inferiormente in ogni periodo considerato. La stabilità fa riferimento al quinto livello del Capability Maturity Model Integration (**CMMI**).

CMMI è un approccio focalizzato sul miglioramento dei processi il cui obiettivo è di aiutare un'organizzazione a migliorare le sue prestazioni.  Il CMMI può essere usato per guidare il miglioramento dei processi all'interno di un progetto, una divisione o un'intera organizzazione, per questo studio viene presa in considerazione il miglioramento del progetto. Il quinto livello di CMMI prevede un continuo miglioramento e quindi viene fatta un’analisi sui bug per controllare la loro stabilità così da poter prevenire eventuali problemi.

## 2.2 Analisi della stabilità

L’analisi effettuata per lo studio della stabilità prevede una rappresentazione grafica temporale del numero di Bug Fixed di Jira in ogni mese da quando è stato iniziato il progetto. Come detto precedentemente il progetto del caso di studio è **Apache Parquet**.

I Bug Fixed sono stati estratti facendo una query verso Jira e prendendo tutti i bug con status closed o resolved e come campo resolution il valore fixed.

Per calcolare il numero di Bug Fixed si procede prendendo per ogni bug fixed la commit più recente temporalmente che contiene l’id del bug nel commento, si procede sommando tutte le commit dello stesso mese. Da notare che se eventualmente una commit dovesse contenere nel suo commento o titolo più di un bug allora ovviamente verranno presi tutti i bug trovati.

Per quanto concerne il progetto considerato, la finestra temporale va da Giugno 2014 ad Aprile 2020.

Le commit sono state raccolte sul branch di default della repository “parquet-mr”.

Il grafico della stabilità contiene espresso come serie temporale del numero di bug fixed per mese contiene 3 rette orizzontali che riguardano :

* Media aritmetica della serie
  + *3.27*
* Upper bound della serie temporale:
  + *average + 3 \* devStd = 14.13*
* Lower bound della serie temporale:
  + *max(0, average - 3 \* devStd) = 0*
  + *average - 3 \* devStd = -6.46, non ha senso avere una misura negativa se parliamo di quantità >= 0 quindi si è scelto di calcolare il max con 0.*

Figura Process Control Chart

Come si evince dal grafico (Figura 1) l’unico mese in cui il numero di bug è superiore all’upper bound riguarda luglio 2015 con 16 bug. Il motivo di tale picco riguarda un incremento sostanziale della produttività in quel periodo, infatti si riscontra che è stata rilasciata una nuova versione proprio il 17 luglio 2015 (apache-parquet-1.8.1) e solo quel giorno sono state effettuate 7 commit.



Figura 2 produttività repository parquet-mr

La Figura 2 rappresenta un grafico della frequenza delle varie modifiche di add e delete per settimana. E’ semplice notare che verso il mese di Luglio 2015 esiste un picco di frequenza.

E’ stata inoltre studiata la distribuzione dei dati (Figura 3).

Figura 3 Distribuzione del numero di Fix

Si è potuto notare che la distribuzione si avvicina ad una Chi quadro con pochi gradi di libertà. E’ ragionevole aspettarsi una distribuzione del genere siccome il numero di fixed bug è strettamente positivo. Dallo stesso grafico si può inoltre vedere che esistono molti mesi che hanno pochi fixed bug, questo significa che il progetto si sta evolvendo bene e mantiene una certa stabilità intorno ai valori bassi.

## 2.3 Ulteriori analisi

Informazioni raccolte:

1. Altri reposirory git trovati

Dallo studio dei bug del progetto PARQUET di Jira, è stato possibile notare che alcuni bug sono associati ad altri repository git oltre a quello ufficiale “parquet-mr”. Sono stati identificati 2 ulteriori repository :

* arrow
* arrow-testing

1. Numero commit e bug

Sono state poi ricavate le seguenti informazioni :

* il numero di commit che hanno associato un id di un bug fixed “PARQUET-####”
* il numero di commit senza bug fixed
* il numero di commit che presentano id di altri progetti
* numero bug con commit associata
* numero bug senza commit associata

1. Branch

Infine sono stati confrontati i progetti prendendo solo il branch di default e i progetti con tutti branch.

Per studiare queste informazioni è stata studiata la stabilità in 4 scenari :

1. Repository parquet-mr, branch di default
2. Repository parquet-mr, 35 branch coinvolti
3. Bug dei 3 repository con branch di default
   * parquet-mr
   * arrow, 5 branch
   * arrow-testing, 2 branch
4. Bug dei 3 repository con tutti i branch

### 2.3.1 Parquet-mr default branch

Il default branch del repository principale è stato studiato nel capitolo precedente. In questo capitolo vengono aggiunte le informazioni riguardanti il numero di commit e di bug:

Figura 4 Conteggio commits e bugs per jira parquet da repository parquet-mr github

Commits

|  |  |  |
| --- | --- | --- |
| Numero commit id bug PARQUET- | 251 | 11.49% |
| Numero commit senza bug | 1632 | 74.69% |
| Numero commit con soli altri ID | 302 | 13.82% |
| Tot commit | 2185 | 100% |
| Bugs |  |  |
| Numero bug con commit | 239 | 42.75% |
| Numero bug senza commit | 320 | 57.25% |

### 2.3.2 Parquet-mr tutti i branch

Figura 5 Conteggio commits e bugs per jira parquet da repository parquet-mr github considerando tutti i branch

Commits

|  |  |  |
| --- | --- | --- |
| Numero commit id bug PARQUET- | 1160 | 2.73% |
| Numero commit senza bug | 40115 | 94.18% |
| Numero commit con soli altri ID | 1317 | 3.09% |
| Tot commit | 42592 | 100% |
| Bugs |  |  |
| Numero bug con commit | 422 | 75.49 % |
| Numero bug senza commit | 137 | 24.51% |

Figura 6 Process Control Chart per jira parquet da repository parquet-mr considerando tutti i branch.

Per quanto concerne l’inclusione delle commit per tutti e 36 i branch di parquet salta subito all’occhio dalla Figura 6 che a Gennaio 2017 c’è un picco di 62 bugs. Anche in questo caso è presente un aumento della produttività probabilmente dovuto al rilascio della versione di parquet 1.8.2 il 19 gennaio 2017.

### 2.3.3 Tutti repo default branch

Figura 7 Conteggio commits e bugs per jira parquet da repository parquet-mr, arrow, arrow-testing github considerando il default branch

Commits

|  |  |  |
| --- | --- | --- |
| Numero commit id bug PARQUET- | 465 | 5.18% |
| Numero commit senza bug | 1923 | 21.39% |
| Numero commit con soli altri ID | 6601 | 73.43% |
| Tot commit | 8989 | 100% |
| Bugs |  |  |
| Numero bug con commit | 419 | 74.96% |
| Numero bug senza commit | 140 | 25.04% |

Figura 8 Process Control Chart per jira parquet da repository parquet-mr, arrow, arrow-testing github considerando il default branch

Dalla Figura 8 è possibile notare che aggiungendo gli altri repository (arrow e arrow-testing) è possibile raggiungere una stabilità maggiore nel caso delle sole commit del branch di default.

### 2.3.4 Tutti repo tutti i branch

Figura 9 Conteggio commits e bugs per Jira parquet da repository parquet-mr, arrow, arrow- testing considerando tutti i branch

Commits

|  |  |  |
| --- | --- | --- |
| Numero commit id bug PARQUET- | 2443 | 3.12% |
| Numero commit senza bug | 41822 | 53.35% |
| Numero commit con soli altri ID | 34126 | 43.53% |
| Tot commit | 78391 | 100% |
| Bugs |  |  |
| Numero bug con commit | 422 | 75.49% |
| Numero bug senza commit | 137 | 24.51% |

Figura 20 Process Control Chart per Jira parquet da repository parquet-mr, arrow, arrow-testing considerando tutti i branch

### 2.3.4 Considerazioni

Sapendo che il branch di default contiene tutte le commit dei branch che sono stati unificati possiamo concludere che le uniche commit aggiunte sono quelle dei branch non ancora unificati al branch di default.

Altre considerazioni sono quelle releative alle commit che hanno un id di ticket di altri progetti Jira, ad esempio alcune commit hanno i seguenti id : HIVE-###,AVRO-###, tali commit sono state escluse nella costruzione del process control chart. Il motivo dell’esistenza di commit legate ad altri bug jira può essere legato al fatto che parquet è un tipo di formato particolare orientato alle colonne che serve per ottimizzare le operazioni di lettura e scrittura effettuate dai framework di big data e quindi deve interfacciarsi con progetti come Hive ( fornisce la possibilità di effettuare query sql su dati non strutturati ) e Avro ( altro formato di dati binario basato su json ), tali progetti fanno parte dell’ecosistema hadoop ossia un insieme di progetti che risolvono problemi di big data e si interfacciano tra loro.

Non è stato studiato il process control chart dei 3 repository insieme poiché il numero dei bug di

## 2.4 Dettagli implementativi

Il programma parte dalla classe Main e continua eseguendo il Controller che richiedere i dati attraverso chiamate RESTfull http a Jira per i bug e a Github per le commit. Le classi parserGithub e parserJira consentono di leggere il Json ricevuto dalle chiamate RESTfull. Per quanto concerne Jira la query utilizzata consente di raccogliere tutti i ticket chiusi o risolti filtrati per attributo “fixed bug”. Mentre da github vengono raccolte tutte le commit con relativa data e testo.

Per conoscere quali commit sono associate ad un determinato id Parquet- è stata utilizzata un’espressione regolare che controlla il matching esatto di ogni id di Jira nel titolo e commento di ogni commit.

L’espressione regolare utilizzata è la seguente:

* “IDBug” + \D| + ”IDBug” + \b

Quindi applicando tale espressione al corpo della commit e sostituendo a IDBug ogni bug di Jira è possibile verificare il matching esatto.

In particolare:

* \D significa che non devono esserci caratteri altri caratteri numerici dopo l’id del bug nel commento della commit,
* | simbolo di or per catturare il caso in cui il carattere successivo è il terminatore di stringhe
* \b permette appunto di controllare se il ticket è boundary, cioè se non sono presenti caratteri dopo

La scelta di utilizzare le commit di github associate ai bug è stata forzata dal fatto che Jira non ha una data di risoluzione dei bug, in particolare il campo resolutionDate di Jira è un campo che viene riempito dagli sviluppatori durante l’apertura del bug ed è soltato una previsione del tempo che serve per risolvere il bug, per tanto non è affidabile. L’unico svantaggio riscontrato incrociando bug e commit è che non tutte le commit sono state riferite al bug di interesse, quindi è possibile trovare dei bug scoperti senza commit e che quindi non rientrano nel conteggio finale. Fortunatamente soltanto circa il 30% di bug come mostrato precedentemente non ha associato nessuna commit.

Conclusa la fase di raccoglimento dei dati, la classe ControllerFirstDelivery si occupa di contare il numero di commit/bug in un mese per poi stampare la lista ottenuta su un file.

I grafici con i relativi calcoli inerenti all’upper bound, lower bound e media sono stati effettuati su Microsoft Excell.

## 2.5 Conclusioni

Possiamo concludere che la stabilità dei progetti è verificata. Possiamo evincere dai grafici che durante la prima parte del progetto esso è un po' meno stabile, mentre avvicinandoci al 2020 i valori sono più vicini alla media. Questo è un chiaro sinonimo di maturità del progetto.

# Capitolo3 Delivery2

## 3.1 Descrizione del problema

Il problema da risolvere è quello di confrontare tecniche e modelli di machine learning al fine di studiare quale combinazione riesce ad effettuare una previsione migliore su quali sono le classi Java definite Buggy.

Tale studio è stato applicato a due progetti in maniera distinta:

* Apache Bookkeeper
* Apache Syncope

Per poter addestrare un modello di machine learning e ricavarne una previsione è necessario costruire un dataset.

Un dataset è una raccolta di dati rappresentata sotto forma di tabella. Ogni colonna rappresenta una particolare feature cioè una caratteristica il cui valore condiziona il risultato che vogliamo predire. Ogni feature ha un dominio di possibili valori.

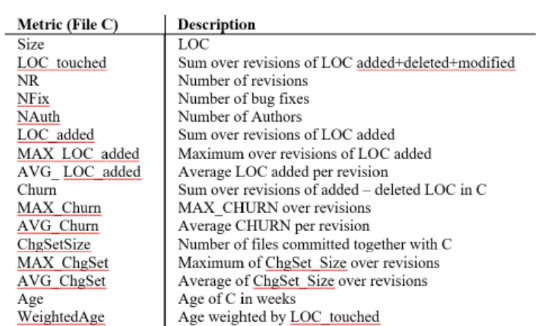
Nello studio effettuato la previsione viene effettuata su una variabile di tipo binario, ossia una variabile che può assumere solamente due valori. Tale variabile binaria è la bugginess, ovvero la difettosità di una classe Java appartenente al progetto.

I modelli considerati sono di tipo Supervised ciò significa che per funzionare devono avere in input un set di dati di addestramento (o training set) già classificati, cioè che hanno associate delle etichette (o label) con il nome della classe di appartenenza. In questo modo il modello potrà estrarre dal training set una funzione complicata a piacere. Questa funziona servirà al modello per valutare la classe di appartenenza dei nuovi dati mai visti non ancora etichettati. L’insieme dei dati non ancora etichettati prende il nome di Test Set.

## Costruzione del dataset

Il tema della predizione dei difetti (defect prediction) è di interesse per molte aziende poiché è possibile prevedere quanto effort impiegare su un progetto in base al numero di bug presenti. Nella letteratura sono stati proposti molti approcci che effettuano previsioni basandosi su differenti metriche. Le metriche prese in considerazione nello studio attuale suppongono che i file difettosi sono quelli che sono stati modificati più frequentemente.

Tabella Metriche per file



La Tabella 1 mostra l’elenco delle metriche che sono state utilizzate come input al modello di previsione.

### Dettagli sul calcolo delle metriche

Le metriche precedentemente elencate sono calcolate leggendo le informazioni delle commit tra una release e la successiva per ogni classe java. Questo significa che per ogni release del progetto si ha un output di tutti i file presenti nella release con le relative metriche calcolate specificatamente in quella release.

Si è scelto di calcolare le metriche release per release azzerandole di volta in volta e non in maniera incrementale per l’intero progetto perché si è constatato attraverso altri studi pregressi di ricerca che il modello avente in input un dataset del genere si comporta meglio e restituisce risutati di accuratezza migliori. Un altro motivo di tale scelta è che l’informazione temporale è già inclusa nell’age della classe e quindi sarebbe ridondante calcolare le metriche in maniera incrementale.

Una metrica in particolare che merita di essere commentata è la WeightedAge, si è scelto di calcolarla secondo l’approccio delle metriche MOSER:

Al termine di ogni release le metriche vengono inizializzate a 0 tranne le due metriche di Age.

Le metriche vengono calcolate per ogni tipo di commit:

* Rename: viene mantenuto il vecchio file con le relative metriche ma ne viene cambiato il filename
* Copy: gestito come una delete e poi una create
* Modify: aggiorno tutte le metriche
* Create: inizializzo il file con le metriche
* Delete: elimino il file e le metriche annesse nella release attuale

### Calcolo del LifeCycle

Una volta concluso il calcolo delle metriche informative per la difettosità della classe si passa al calcolo della bugginess per ogni classe considerata.

Tale stima è stata effettuata attraverso il metodo Proportion. Tale metodo ha come obiettivo quello di impostare una release al valore Affected o no, ossia dire se una certa release è affetta da un certo difetto. Proportion utilizza i seguenti stati del ticket:

* **Injected Version:** release affetta più vecchia. Se da Jira sono presenti le Affected Version allora l’injected è la prima versione di queste altrimenti viene calcolata tornando indietro con Proportion
* **Opening Version:** release dopo l’apertura del ticket
* **Fixed Version:** release in cui è stato risolto o chiuso il bug, viene calcolata prendendo la release dopo l’ultima commit del bug
* **Affected Version:** versioni affette dall’Injected alla Fixed esclusa. Se presenti vengono prese quelle nel campo di Jira altrimenti viene calcolata prendendo le versioni da Injected a Fixed.

Il metodo Proportion suppone che per ogni bug il rapporto tra FV-IV e FV-OV rimanga più o meno costante.

Dalla formula (2) si ricava che:

Con la formula (3) è possibile calcolare quale sia la release Injected e da qui impostare ad affected tutte le versioni fino alla Fixed version esclusa.

Bisogna far notare che per evitare problemi di snoring, cioè evitare che esistano dei bug dormienti che non vengono rilevati dal metodo nelle versioni più recenti, si è scelto di utilizzare solamente il 50% delle release più vecchie.

### Bug da escludere

Durante lo studio delle release affette è possibile incontrare alcuni casi particolare che sono stati esclusi dall’analisi:

* Bug senza fixed version
* Bug con open versione maggiore o uguale di fixed version
* Bug senza open version

Nel caso in cui Injected version stesse dopo opening version viene applicato Simple method e si porta Injected sull’opening version.

### Tecniche per calcolare P

Tra i metodi studiati per calcolare P si è scelto di applicarne 2:

* moving window : si stima P facendo una media degli ultimi P suill’ 1% dei bug precedenti con AV di Jira
* increment: si stima P facendo una media tra tutti i P dei bug precedenti con AV di Jira

Un problema riscontrato in questa fase è quello di calcolare P all’inizio quando non si hanno bug oppure non ho ancora accumulato 1%, in questo caso specifico si è scelto di utilizzare Simple Method e quindi impostare Injected Version all’Opening Version.

Vale la pena evidenziare che l’1% dei bug nel caso del progetto di Bookkeeper è inferiore ad 1. Si è deciso quindi di settare l’1% ad 1 in casi come questo.

### Classi buggy

Una volta calcolate le metriche e trovate le release affette per ogni bug si conclude la costruzione del dataset settando la categoria della classe a buggy o non buggy.

Per settare la classe come buggy bisogna trovarla in una release affetta di un particolare bug e poi trovare almeno un’operazione di modify su di essa. Si setta quindi, la classe come buggy dall’injected version alla fixed version esclusa.

Una problematica riscontrata in questa fase riguarda i file che hanno cambiato nome durante le commit di rename. Quando una classe viene settata come buggy bisogna settare la classe buggy in tutte le release affette dall’injected alla fixed e quindi bisogna settare buggy anche le la classe con il vecchio nome. Per risolvere tale problematica è stato salvato il vecchio nome in un campo dell’entità File nel progetto Java e si vanno a cercare ricorsivamente tutti i vecchi riferimenti.

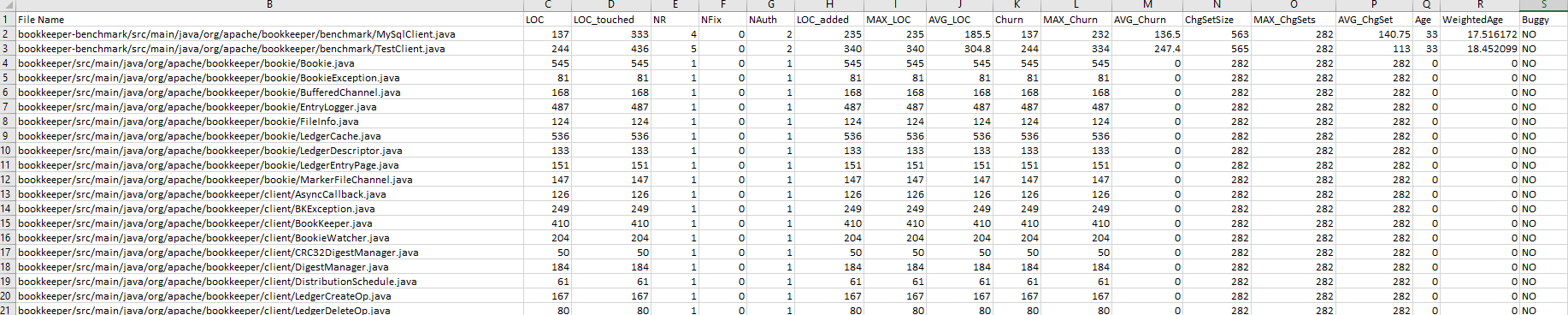


Figura 21 estratto del dataset prodotto del progetto Apache Bookkeeper

## Dettagli implementativi

Per la seconda delivery si è scelto di ricavare le commit attraverso la libreria Jgit poiché le chiamate RESTfull http sono di numero limitato per fascia oraria. Con tale libreria viene effettuato il download del progetto ed è possibile scorrere tutte le commit esistenti.

### Scelta del branch

Un altro aspetto che vale la pena menzionare riguarda i branch del progetto Git.

La scelta fatta è quella di considerare solamente il ramo principale del branch Master e quindi escludere una buona parte dei commit che risiedono nei vari rami mergiati con il branch Master.

Tale scelta è motivo della problematica che fa riferimento alla differenza cronologica commit per commit partendo dal nodo di merge tra più branch.

Quindi se si fossero analizzate anche le commit di percorsi dovuti al merge con altri branch si sarebbe dovuto partire dal nodo di merge e tornare indietro analizzando la differenza tra le commit in ordine cronologico fino ad arrivare al nodo genitore. Il problema nasce quando alla fine della release esistono branch che non sono ancora stati mergiati con il master e quindi l’analisi impattarebbe su più release. Impattando su più release diventa di difficile gestione l’analisi delle metriche relative ad una sola release.

Un'altra scelta possibile sarebbe stata quella di escludere solamente i branch che hanno durata maggiore ad una release e mantenere gli altri, ma tale implementazione avrebbe richiesto più tempo del necessario.

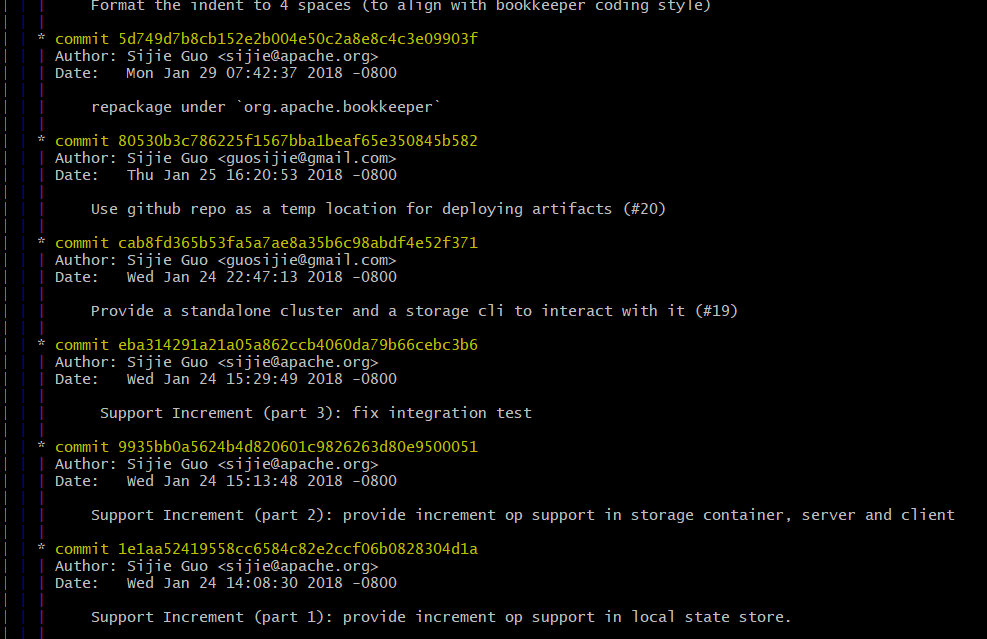


Figura 22 git log commit

Il ramo di commit considerato nel progetto è quello di colore verde (Figura 21). Come spiegato precedentemente i rami di colore blu e viola potrebbero durare diverse feature e quindi sarebbe complicato attendere il nodo di merge e poi analizzare a ritroso tutte le commit tra più release.

Tuttavia la percentuale di commit escluse risulta comunque bassa.

Un’altra peculiarità riguarda la presenza di branch orfani presenti ad esempio in bookkeeper.

I branch orfani sono dei branch particolari che non risultano essere figli di nessuna commit esistente, essi iniziano da una differente radice di commit. I branch orfani sono privi di storia pregressa e possono essere creati da Git con il flag “—orphan”. Solitamente vengono creati per scopi di testing oppure per sviluppare una nuova feature del tutto scorrelata dal resto dei branch.

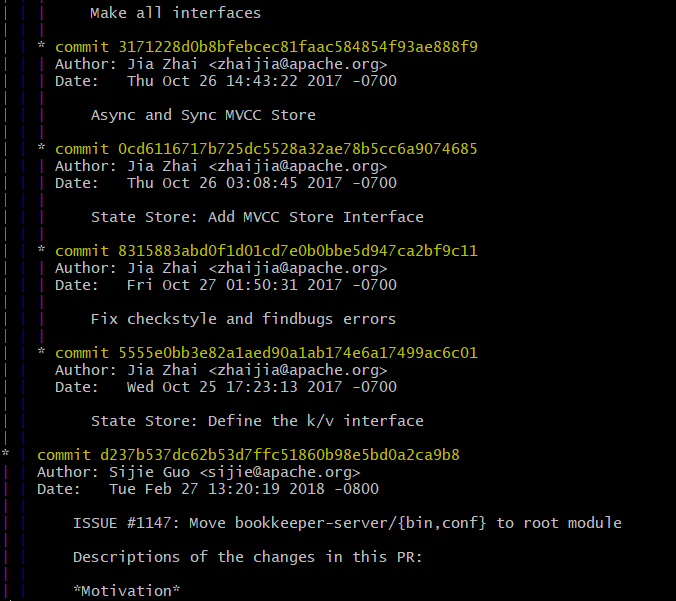


Figura 23 estratto dal git log del progetto di bookkeeper, commit 5555e0bb3e.. è l’inizio di un branch orfano

## weka machine learning

La parte di progetto inerente alla predizione è stata sviluppata in Java utilizzando la libreria di machine learning Weka.

Il modello prende in input il dataset creato e lo utilizza per addestrare un modello.

Sono state quindi confrontate le seguenti tecniche e i seguenti modelli:

|  |  |
| --- | --- |
| Tecnica di valutazione | Walk Forward |
| Modelli | * Naive Bayes * Random Forest * IBK con tuning su K (1,3,9,19) * Logistic regression |
| Feature selection | * No feature selection * Best First * Wrapper con Naive Bayes * Principal component analysis(PCA) |
| Balancing | * No sampling * Undersampling * Oversampling * SMOTE |

Tali tecniche e modelli sono state applicate ai seguenti dataset:

* Dataset Apache Bookkeeper con metodo moving window
* Dataset Apache Bookkeeper con metodo increment
* Dataset Apache Syncope con metodo moving window
* Dataset Apache Syncope con metodo increment

Al fine di effettuare un confronto tra i modelli sono state scelte le seguenti metriche:

* AUC
* KAPPA
* Precision
* Recall
* FMeasure

### Risultati: Bookkeeper – moving window ml

Al fine di rispondere alle domande richieste, cioè per quali tecniche di feature selection o balancing si ottiene un aumento delle prestazioni dei classificatori, sono stati prodotti i seguenti grafici che aiutano a rispondere a tali domande. Da notare che è stata utilizzata la stessa scala di valori anche tra metriche diverse per avere un’immagine comparabile delle grandezze.

### Risultati: Bookkeeper – moving window ml – modello vs feature selection



Figura 24 Classificatori vs feature selection no sampling



Figura 25 Classificatori vs feature selection oversampling

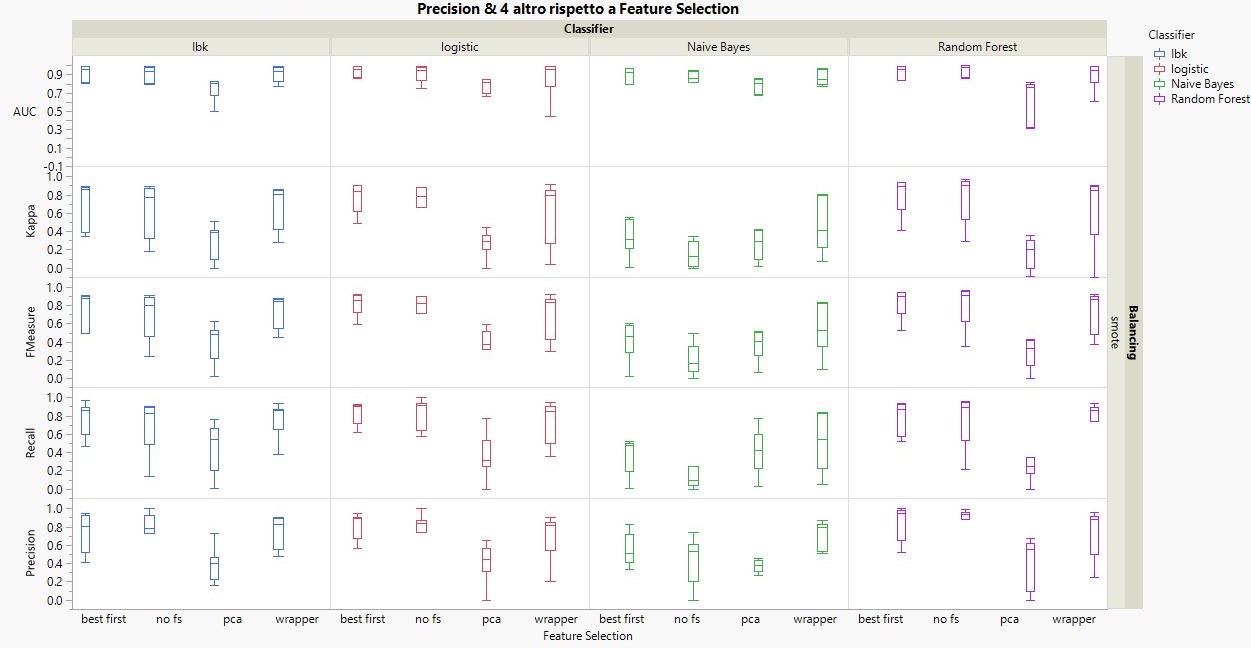


Figura 26 Classificatori vs feature selection smote



Figura 27 Classificatori vs feature selection undersampling

Grafico valore medio Tuning K

Grafici con percentuale bug per release

Box plot con mediana tra release

Box plot re delle visualizzazioni perche ha la mediana e la dispersione

SECONDA DOMANDA SUL PROGETTO: è POSSIBILE SELEZIONARE LA TECNICA MIGLIORE?

TRAINING VALIDATION

TRAINING VALIDATION

TRAINIG VALIDATION

MODELLO- TUNING- TEST SET

PER OGNI WALK FORWARD CALCOLO LA MEDIANA DEI RUN

# Riferimenti sul Web

1. <https://docs.atlassian.com/fisheye-crucible/latest/wadl/crucible.html#d2e221>
2. <https://docs.atlassian.com/software/jira/docs/api/REST/7.6.1/#api/2/project-getAllProjects>
3. <https://developer.atlassian.com/cloud/jira/platform/rest/v3/#version>
4. <https://confluence.atlassian.com/display/JIRASOFTWARESERVER083/Advanced+searching>
5. <https://asankhaya.github.io/pdf/automated-identification-of-security-issues-from-commit-messages-and-bug-reports.pdf>
6. <https://github.com/apache/parquet-mr/tags>