



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Ingegneria del Software**

Studio di tecniche di predizione 'Just-in-Time' di commit difettosi nello sviluppo software in contesti di Continuous Integration

Anno Accademico 2020-2021

Candidato:

Giuseppe Riccio

matr. N46004297

Relatore:

Prof. Ing. Roberto Pietrantuono

*Alla mia famiglia ed ai miei amici
che mi hanno sostenuto ed
incoraggiato durante questo
fantastico percorso*

Indice

Indice.....	III
Introduzione	4
Glossario dei termini	5
Capitolo 1: Sviluppo del software.....	6
1.1 Metodologie agili	6
1.1.1 eXtreme Programming	8
1.1.2 DevOps.....	9
Capitolo 2: Continuous practices	11
2.1 Continuous Integration / Continuous Delivery / Continuous Deployment.....	12
2.1.1 Versioning Control System : Git e GitHub.....	13
2.1.2 Come usare GitHub a supporto della Continuous Integration	15
2.2 Problemi legati all'uso delle Continuous Practices.....	16
Capitolo 3: Tecniche di predizione dei commit difettosi	18
3.1 Just-in-Time Prediction.....	18
3.2 Processo di costruzione e validazione del modello predittivo	20
3.2.1 Estrazione delle metriche tramite Commit Guru	24
3.2.2 Costruzione del modello con Weka	29
3.2.3 Misurazione delle performance delle metriche	31
Capitolo 4: Caso di studio: Immuni App, Android vs. iOS	35
4.1 Progettazione del caso di studio.....	36
4.1.1 Estrazione delle metriche e definizione dei parametri di classificazione	36
4.1.2 Costruzione del modello a partire dal dataset	42
4.1.3 Validazione del modello	42
4.1.4 Risultati ed osservazioni	42
Conclusioni	43
Bibliografia	44

Introduzione

All'interno di questa tesi di Laurea verranno trattate le tecniche di predizione di commit difettosi nell'ambito di sviluppo del software con modello Continuous Integration, con il supporto di strumenti per il controllo di versione come GitHub. In particolare, affronteremo attraverso l'uso della metodologia del Just-in-Time Prediction, l'estrazione delle metriche a partire da un log di commit e la creazione di un file .csv. Ogni commit viene etichettato come difettoso o meno, questi dati rappresentano il training set dal quale costruire un modello predittivo.

Poi ci occuperemo di validare questo modello sottoponendogli dei commit non etichettati, quindi, verificheremo le predizioni date dal modello al fine di valutare la sua precisione e correttezza nei risultati.

A questo proposito, tratteremo un caso di studio particolare, al fine di ottenere dei risultati rilevanti su cui poter fare alcune osservazioni di particolare rilevanza.

Glossario dei termini

- **ECU**: Electronic Control Unit, Capitolo 1: Introduzione pag. 6
- **IT**: Information Technology, Capitolo 1: 1.1.2 DevOps pag. 9
- **URL**: Uniform Resource Locator, Capitolo 3: 3.2.1 Estrazione delle metriche tramite Commit Guru pag. 24

Capitolo 1: Sviluppo del software

La componente software nei sistemi odierni ha ormai raggiunto un'importanza molto alta, quasi alla pari se non anche superiore rispetto alla componente hardware. I primi calcolatori programmabili risalgono agli anni '40-'50 usati principalmente in ambito militare e costituiti da software molto semplici. Ai nostri giorni, il software è presente in ogni aspetto della nostra vita, basti pensare alle ECU delle automobili (centraline), agli elettrodomestici, ai sistemi industriali, ai sistemi informativi di aziende e Pubblica Amministrazione, etc.

Ciò ha portato alla definizione di standard e metodologie per lo sviluppo del software, ma come accade per ogni processo industriale insieme alla nascita di nuove tecniche nascono anche alcuni problemi legati ad esempio, a tempi di sviluppo più ampi o costi maggiori di quanto preventivato a fronte di richieste da parte dei clienti sempre più esigenti. Inoltre, il software non è un prodotto “statico” che, una volta completato rimane costante ma, nella maggior parte dei casi, richiede continui aggiornamenti sia di tipo manutentivo che di tipo incrementativo, per aggiungere nuove funzionalità, ciò significa che anche le fasi di sviluppo di un software non devono essere “statiche” ma bensì “dinamiche” al fine di permettere il suo continuo aggiornamento.

1.1 Metodologie agili

Per risolvere i problemi dei modelli tradizionali di sviluppo del software, tra cui ad esempio il modello “a cascata”, negli anni '90 nasce un nuovo movimento che porta alla proposta delle cosiddette *metodologie agili*.

Tali metodologie, presentano come caratteristiche fondamentali quelle di:

- Concentrarsi sul codice, piuttosto che sulla progettazione.
- Basarsi su un approccio iterativo allo sviluppo software, con continui rilasci.
- Rilasciare software funzionante rapidamente, al fine di ricevere feedback rapidi da parte dei clienti.

Il movimento dell'Agile Software Development, formalizzato nel febbraio 2001, propone dei modelli di sviluppo “leggeri” in alternativa alle metodologie tradizionali, la peculiarità di questi modelli è che essi sono adattivi più che predittivi perché non cercano di programmare lo sviluppo nel dettaglio e in modo da soddisfare tutte le specifiche, ma progettano programmi pensati per cambiare nel tempo.

Al fine di rendere lo sviluppo più rapido si cerca di aumentare il coinvolgimento delle persone all'interno del processo produttivo, rendendo quest'ultimo *people-oriented* anziché *process-oriented* tramite l'adattamento del processo alla natura dell'uomo per rendere lo sviluppo software un'attività piacevole per chi lo opera.

Questo tipo di metodologia è particolarmente conveniente in tutti quei progetti caratterizzati da un obiettivo non sufficientemente chiaro ed in cui i requisiti sono poco chiari, instabili e variabili nel tempo. Inoltre, le metodologie agili non hanno una struttura rigida e fissa nel tempo, questo permette di usarle in tutte quelle aziende in cui il processo di sviluppo software è ancora acerbo e/o ignoto, come ad esempio per nuove start-up.

Le caratteristiche spiegate in precedenza sono state riassunte dai membri fondatori del movimento dell'Agile Software Development in un manifesto, in particolare, sono stati stilati 12 principi [\[1\]](#) che ogni metodologia agile deve rispecchiare:

1. La nostra massima priorità è soddisfare il cliente rilasciando software di valore, fin da subito e in maniera continua.
2. Accogliamo i cambiamenti nei requisiti, anche a stadi avanzati dello sviluppo. I processi agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente.
3. Consegniamo frequentemente software funzionante, con cadenza variabile da un paio di settimane a un paio di mesi, preferendo i periodi brevi.

4. Committenti e sviluppatori devono lavorare insieme quotidianamente per tutta la durata del progetto.
5. Fondiamo i progetti su individui motivati. Diamo loro l'ambiente e il supporto di cui hanno bisogno e confidiamo nella loro capacità di portare il lavoro a termine.
6. Una conversazione faccia a faccia è il modo più efficiente e più efficace per comunicare con il team ed all'interno del team.
7. Il software funzionante è il principale metro di misura di progresso.
8. I processi agili promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere indefinitamente un ritmo costante.
9. La continua attenzione all'eccellenza tecnica e alla buona progettazione esaltano l'agilità.
10. La semplicità - l'arte di massimizzare la quantità di lavoro non svolto - è essenziale.
11. Le architetture, i requisiti e la progettazione migliori emergono da team che si auto-organizzano.
12. A intervalli regolari il team riflette su come diventare più efficace, dopodiché regola e adatta il proprio comportamento di conseguenza.

1.1.1 eXtreme Programming

L'eXtreme Programming, o XP, è una delle metodologie agili più diffuse, l'approccio usato in questa metodologia è basato su iterazioni veloci che rilasciano piccoli incrementi delle funzionalità molto frequentemente.

La progettazione è incentrata sull'iterazione attuale, lasciando tutti i dettagli relativi ai requisiti implementati in futuro alle iterazioni successive.

Ad ogni iterazione si effettua un'operazione di *refactoring* che permette di integrare al meglio le nuove funzionalità con quelle già presenti, in questo modo si ottiene un codice in continuo e costante miglioramento.

Questo modello di sviluppo prevede, inoltre, la partecipazione attiva del committente alla

progettazione del prodotto software al fine di ridurre al minimo le possibili interpretazioni errate circa le richieste e le caratteristiche che il sistema deve rispettare.

L'eXtreme Programming, prevede una serie di best practices al fine di applicare al meglio questa metodologia di sviluppo:

- **Planning game:** Determina obiettivo e tempi della prossima release.
- **Small releases:** Rilasciare velocemente un piccolo incremento.
- **Metaphor:** Guida lo sviluppo attraverso una “storia” condivisa che descrive l'intero sistema.
- **Simple design:** Il sistema è concepito nel modo più semplice possibile.
- **Refactoring:** Ristrutturare il sistema senza cambiarne il comportamento.
- **Testing:** Da effettuare costantemente durante lo sviluppo (Test Driven Development, TDD).
- **Pair programming:** il codice è scritto da due programmatori in coppia sulla stessa macchina.
- **Collective ownership:** Chiunque può cambiare qualsiasi parte del codice quando vuole.
- **Continuous integration:** l'integrazione viene effettuata anche più volte al giorno.
- **40-hour week:** È sconsigliato lavorare più di 40 ore la settimana.
- **On-site customer:** Un rappresentante del cliente deve essere sempre a disposizione.
- **Coding standards:** Supportano la comunicazione durante la produzione del codice.

1.1.2 DevOps

Questa metodologia è costituita da un insieme di pratiche per colmare il divario tra la fase di sviluppo (agile) del software e la fase operativa (IT operations).

L'idea di base è che i team di sviluppo e quelli di IT operations, che si preoccupano cioè di garantire il corretto funzionamento del software dopo il rilascio, collaborino durante tutto il ciclo di vita, dalla pianificazione del processo di sviluppo fino alla fase di monitoraggio, creando una sorta di ciclo infinito come mostrato in Figura 1.

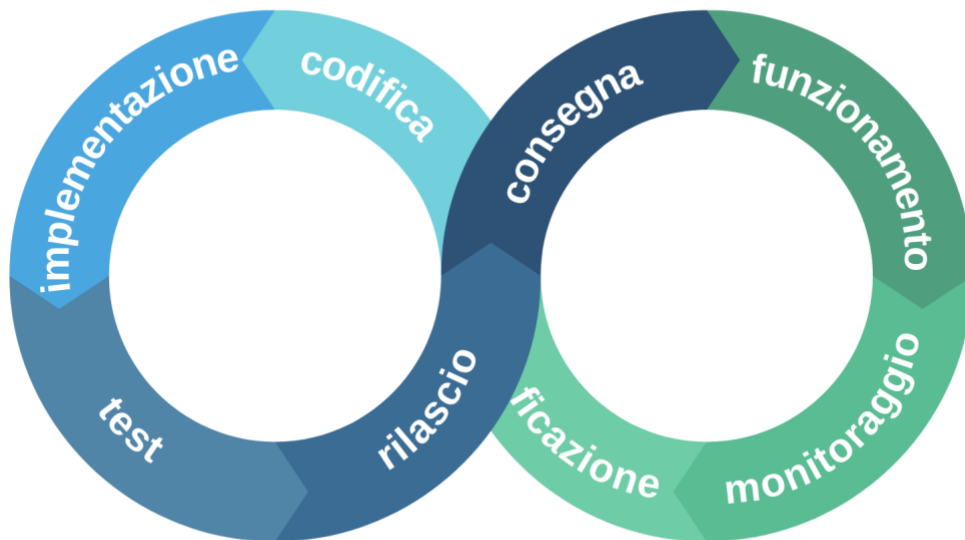


Figura 1: Ciclo DevOps

I principi su cui si basa il DevOps sono:

1. **Systems thinking**, ovvero un sistema in grado di fornire una soluzione ad un problema complesso.
2. **Enfasi sul feedback** dato dalla fase operativa, ovvero dalla fase dopo il rilascio.
3. Cultura di una “**sperimentazione**” **continua** al fine di alleggerire lo sviluppo.
4. **CAMS**:
 - **Culture**: Enfasi sulle persone, comunicazione, cooperazione.
 - **Automation**: Automatic Build, Integrate/Test, Deploy, Monitor.
 - **Measurement**: Supporto alla pianificazione, all'analisi dei trend ed alla qualità del prodotto finale.
 - **Sharing**: Collaborazione, feedback.

Mentre, le best practices DevOps più utilizzate sono:

- **Test automatico** (tramite opportuni framework, ad esempio JUnit).
- **Monitoraggio proattivo**.
- **Continuous Integration/Delivery/Deployment**.
- **Virtualisation/Cloud/Containers**.
- **Processi (Dev e Ops) “visibili”**.
- **Toolchain per l'automazione**.
- **Version control per ogni artefatto** (con l'uso di strumenti, quali GitHub).

Capitolo 2: Continuous practices

Con l'aumento della richiesta di produzione di software, le aziende stanno cercando tecniche semplici ed affidabili per lo sviluppo rapido del software, in questo contesto trovano particolarmente spazio le “*Continuous practices*”, ovvero l'insieme di Continuous Integration (CI), Continuous Delivery (CDE) e Continuous Deployment (CD) [\[2\]](#). Queste pratiche si prefiggono come scopo quello di favorire lo sviluppo ed il rilascio frequente di software senza andare a compromettere la qualità del software stesso, con tecniche di prevenzione da errori che vedremo più avanti.

Mentre, la Continuous Integration si occupa di implementare ed integrare nuove funzionalità nel sistema, anche più volte al giorno, il Continuous Delivery ed il Continuous Deployment si occupano di rilasciare e consegnare in maniera sicura al cliente il software aggiornato.

Quanto, appena detto può essere riassunto nei seguenti tre benefici, che l'applicazione di *Continuous practices* porta con sé:

1. Ricevere più feedback in maniera rapida da parte dei clienti, al fine di supportare lo sviluppo software con continui suggerimenti.
2. Rilasciare subito il prodotto software porta il cliente ad uno stato di maggior soddisfazione, rispetto ad un'attesa più lunga.
3. Grazie alla CD, ed in particolare all'uso di particolari tool la fase di rilascio e monitoraggio di un software in vita vengono totalmente automatizzati con un notevole beneficio economico.

2.1 Continuous Integration / Continuous Delivery / Continuous Deployment



Figura 2: CI/CDE/CD Pipeline

Continuous Integration (CI), in questa fase avviene la codifica del sistema software, ci sono vari team di lavoro ognuno dei quali si occupa di implementare una specifica funzione del software in un linguaggio di programmazione (ad esempio, C++, Java, Python, etc.). Il codice sorgente di ogni funzione viene poi compilato e testato con strumenti completamente automatizzati, come si può evincere dalla Figura 3, infatti abbiamo tool quali quelli di *Code Management and Analysis* e quelli di *Testing* che ci permettono di verificare il corretto funzionamento del software. I test sono effettuati sia sul rilascio attuale sia sul sistema integrato con il nuovo rilascio, quest'ultimo viene spesso chiamato *Regression Test*, proprio perché verifica che la nuova release non aggiunga dei bug che prima non esistevano, ovvero garantisce che il nuovo sistema non regredisca. Oltre ai test funzionali, si effettuano anche dei test di qualità, come ad esempio, in caso di software real-time si effettuano dei test sulla latenza dei risultati al fine di garantire che il programma produca i suoi risultati con un ritardo prefissato e non troppo grande.

Continuous Delivery (CDE), una volta che il software ha passato i test funzionali e di qualità, esso può essere rilasciato su un repository, ovvero una directory (locale e/o distribuita), al cui interno sono presenti tutti i file del sistema software. Questo passaggio viene spesso eseguito manualmente nel caso di CDE, tuttavia anche questa fase fa largamente uso di tool automatizzati, quali ad esempio, quelli di *Versioning Control System* come descritti in Figura 3. La fase di Delivery rappresenta uno step intermedio necessario per ridurre i rischi della fase di Deployment, nonché permettere una maggiore collaborazione tra i vari team di lavoro.

Continuous Deployment (CD), questa fase rappresenta il passo finale attraverso cui avviene il rilascio sul mercato del prodotto software ed allo stesso tempo, essa permette il monitoraggio del software stesso per garantirne il corretto funzionamento durante la sua vita operativa. Gli strumenti a supporto di questa fase prevedono l'uso di *CD Server*, che permettono di distribuire il software in maniera sicura ed affidabile attraverso delle soluzioni “commerciali”, con costi notevolmente inferiori rispetto a soluzioni “proprietarie”.

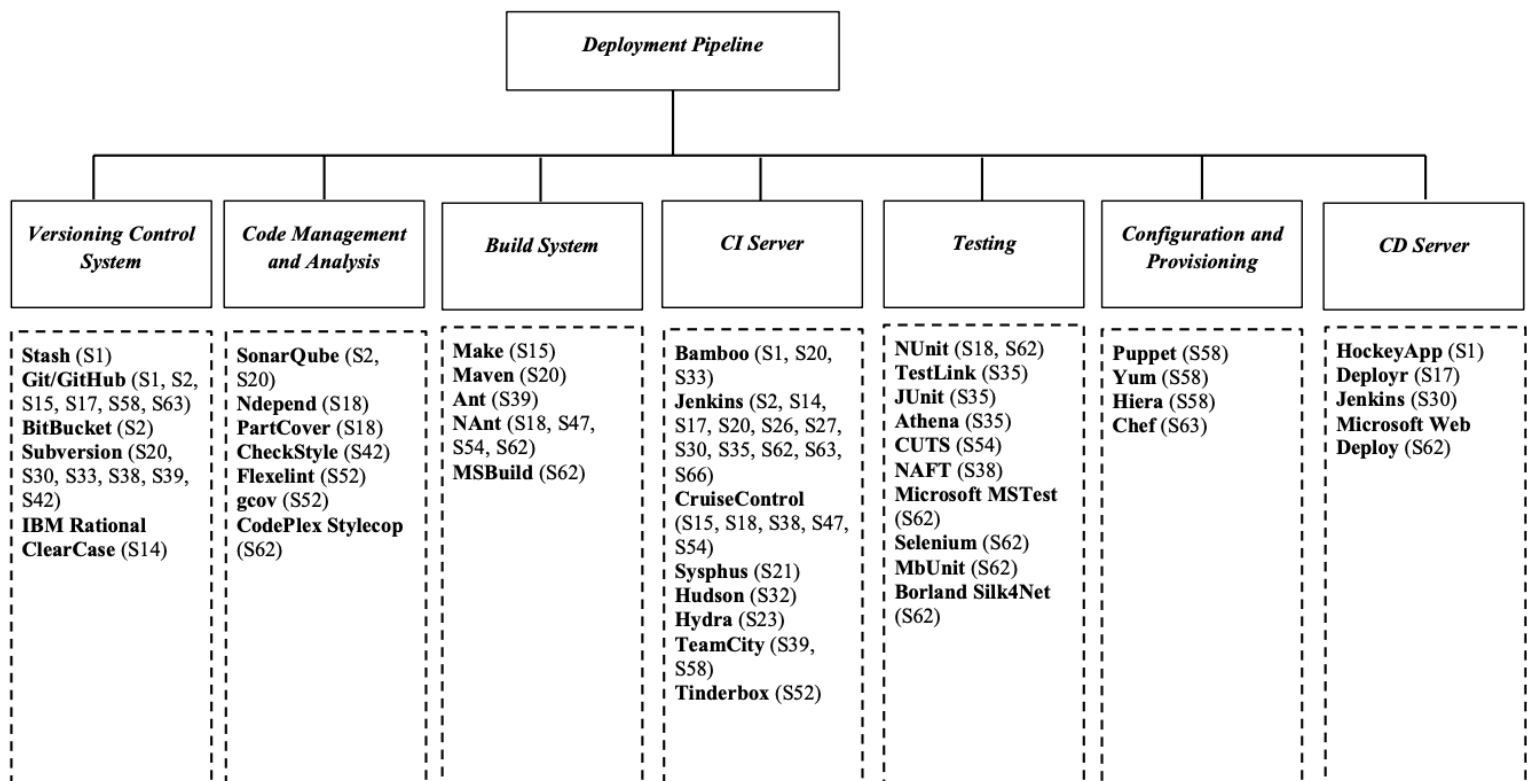


Figura 3: Strumenti a supporto della CI/CDE/CD Pipeline

2.1.1 Versioning Control System : Git e GitHub

Anche se spesso usati insieme in maniera del tutto trasparente all'utente finale, Git e GitHub sono due strumenti separati tra loro. Il primo è un software di controllo di versione per sviluppatori, questo vuol dire che tramite il suo uso è possibile salvare non solo diversi file che compongono un progetto, ma anche le diverse versioni di un file stesso. Questa funzionalità è molto utile nel momento in cui l'ultima versione di un file, ad esempio, viene corrotta o persa irreparabilmente, infatti in questo modo è possibile ripartire

dall'ultima versione salvata. Ma cosa ancora più importante nell'ambito dello sviluppo software, questo sistema di controllo di versione permette di tenere traccia di tutte le versioni di un prodotto software con la possibilità di effettuare delle comparative tra differenti versioni alla ricerca di eventuali difetti introdotti.

Git permette anche di creare dei *branch*, ovvero dei rami, che a partire da una radice comune (*master*) si sviluppano in maniera indipendente ed in parallelo tra di loro, questo meccanismo risulta particolarmente utile quando il software è composto da funzionalità che operano indipendentemente l'una dalle altre e permette, quindi, di lavorarci isolatamente riducendo anche il rischio di interferenze.

Con Git, inoltre, è disponibile un meccanismo di *Push* e *Pull*, con cui uno sviluppatore può inoltrare/ricevere le modifiche effettuate su un file verso/da altri sviluppatori, in modo da rendere più facile la collaborazione ed aumentare allo stesso tempo la produttività.

A supporto proprio di quest'ultima funzione ci viene in aiuto GitHub, una piattaforma che permette di creare dei repository di codice condiviso con altri utenti e basata su un'architettura cloud. Oltre a fornire un servizio di storage, GitHub fornisce tutta una serie di strumenti che permettono di gestire al meglio il lavoro di gruppo definendo ruoli e permessi per ciascun collaboratore. Le principali azioni che si possono intraprendere su GitHub sono tre:

- **Fork:** con questa operazione si può copiare il codice da un repository e modificarlo.
- **Pull:** permette di sincronizzare il codice modificato in locale sul repository al quale il file è collegato.
- **Merge:** permette di unire le modifiche fatte da due o più persone su un codice in un unico file.

A questo punto, risulta chiaro che Git viene usato in locale mentre GitHub è usato in remoto, integrare questi due strumenti è piuttosto semplice, infatti, i repository possono essere clonate in locale con il comando *git clone urlrepository*, dal momento in cui facciamo questo tramite un terminale Linux/MacOS o tramite la bash di Git per Windows, potremo usare i comandi *git pull/push* per sincronizzare il codice scritto in locale con quello presente in remoto su GitHub.

2.1.2 Come usare GitHub a supporto della Continuous Integration

GitHub permette di gestire la build ed il test di un nuovo rilascio all'interno di un repository, infatti, tramite un'opportuna configurazione è possibile automatizzare queste attività ogni qualvolta si integri una modifica al codice.

In particolare, è possibile settare tramite la sezione “*Actions*”, Figura 4, un workflow che permette di eseguire la build ed il test del codice all'interno del repository per verificarne la corretta integrazione. Questo Continuous Integration workflow si innesca al verificarsi di un evento (*trigger*), questo evento può essere ad esempio dato dal comando “*git push*”, una volta innescato, il workflow inizia la build ed il test del software presente nel repository, alla fine di queste operazioni i risultati vengono posti all'interno di una “*pull request*”. In caso di esito positivo della build e del test, la pull request viene accettata e le modifiche vengono unite con un *merge* al repository, in caso di esito negativo vengono segnalati uno o più errori che devono essere risolti all'interno del codice.

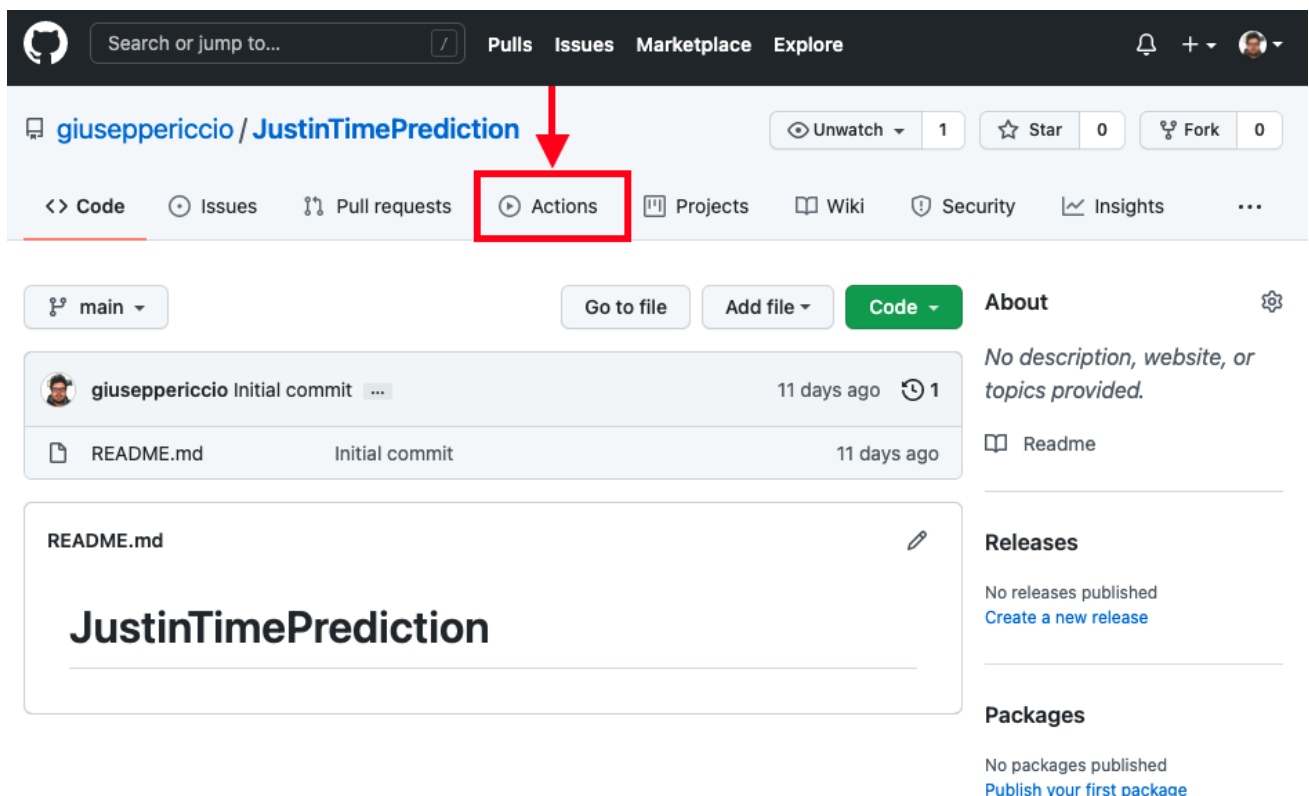


Figura 4: GitHub Actions

I workflow forniscono il supporto ad una grande varietà di linguaggi e per ognuno di essi fornisce alcuni modelli di workflow di base da poter usare, al cui interno sono specificati i trigger d'attivazione ed i test da effettuare, Figura 5.

Continuous integration workflows

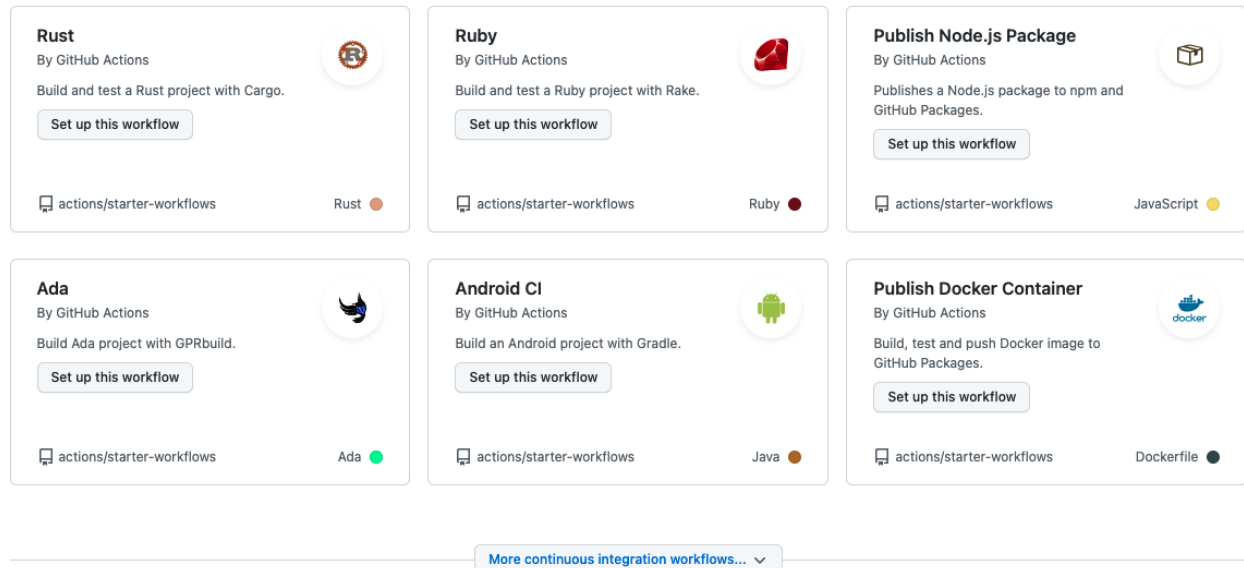


Figura 5: Continuous Integration workflows su GitHub

2.2 Problemi legati all'uso delle Continuous Practices

L'uso delle Continuous Practices, se da una parte permette di velocizzare e semplificare il processo di sviluppo, dall'altra parte aumenta la probabilità di introdurre dei commit difettosi che corrompono il sistema.

L'introduzione di bug all'interno del sistema è una situazione frequente quando l'integrazione di nuove funzionalità avviene rapidamente e diverse volte al giorno, cosa che accade proprio nella Continuous Integration.

Inoltre, essendo che con l'uso di GitHub su uno stesso repository possono lavorare diversi team di lavoro contemporaneamente, la condizione di fallimento può diventare ancora più grave dato che possono essere fatti dei commit su build rotte, in questo modo si accumulano sempre più errori sul software.

Esistono diversi accorgimenti [3] che permettono di ridurre il verificarsi di questi problemi, che devono essere tenuti durante il processo di sviluppo del software

nell'ambito Continuous Integration. Il primo accorgimento è quello di ***non andare a casa se la build è rotta***, infatti, è più semplice correggere il commit difettoso subito dopo averlo individuato perché può essere ridotto il numero di linee di codice che avrebbero potuto corrompere la build limitandoci a quelle dove è stata messa mano recentemente.

Se invece, passano alcuni giorni prima di correggere il commit difettoso diventa più difficile capire dove sia l'errore, perché magari nel frattempo anche altri sviluppatori hanno messo mano al codice. Al fine di evitare queste situazioni, quindi, è preferibile non effettuare commit in prossimità della fine della giornata lavorativa, ma effettuarli il prima possibile durante l'orario di lavoro.

Talvolta, i bug introdotti in un sistema sono così ampi e difficili da risolvere che l'unico modo per tornare ad una situazione di regolare funzionamento del sistema software è quello di **ripristinare l'ultima versione del sistema** prima che venisse effettuato il commit difettoso. Successivamente, si implementano di nuovo le funzionalità eliminate con il commit corrotto, prestando questa volta particolare attenzione nella scrittura del codice. Questa soluzione viene presa anche quando il sistema è ***safety critical***, ovvero ha un tempo massimo entro il quale deve essere risolto il problema, perché un suo prolungato malfunzionamento provoca gravi danni sia a cose che a persone.

Anche se questi accorgimenti sono molto utili nella gestione di commit difettosi, è meglio prevenire delle situazioni del genere perché correggere un commit difettoso integrato nel sistema software presenta un costo sicuramente maggiore di risolvere lo stesso errore isolatamente.

Capitolo 3: Tecniche di predizione dei commit difettosi

Al fine di evitare le condizioni di fallimento enunciate nel [paragrafo 2.2](#) del precedente capitolo, sono state introdotte diverse tecniche di predizione che permettono di individuare in maniera preventiva dei commit difettosi, ancora prima che questi siano integrati nel sistema software.

Le tecniche di predizione tradizionali hanno come obiettivo quello di predire quali file sono difettosi, ovvero introducono bug che intaccano il corretto funzionamento del programma. Tuttavia, risulta dispendioso, nonché inutile controllare ogni qualvolta ci sia un cambiamento nel codice l'intero sistema e quindi, tutti i file che lo compongono.

Per questo sono nate tecniche, come la Just-in-Time Prediction che effettuano la predizione di difetti nei singoli commit, invece, che in un intero file. Per fare ciò, la JIT si focalizza su una serie di caratteristiche (*metriche*) che contraddistinguono un commit in base alle modifiche che esso apporta al sistema.

Inoltre, al fine di valutare le tecniche di predizione, studiate in questo elaborato, si farà uso di alcuni indici di performance che permettono di valutare la bontà di un modello predittivo e dei suoi risultati.

3.1 Just-in-Time Prediction

La Just-in-Time Prediction [\[4\]](#) prevede, come anticipato prima, l'uso di un insieme di metriche che permettono di classificare i commit di un progetto software sulla base di alcune caratteristiche del codice che stanno modificando. In particolare, in alcuni casi vengono prese come riferimento le linee di codice modificate, aggiunte e/o eliminate dal

commit preso in esame, queste metriche vengono dette *Change Metrics* e possono essere raggruppate nella Tabella 1.

Tabella 1: Change Metrics

Classe	Nome	Definizione
Diffusion Metrics	NS	Numero di sottosistemi modificati
	ND	Numero di directory modificate
	NF	Numero di file modificati
	Entropy	Distribuzione del codice modificato lungo il file
Size Metrics	LA	Numero di linee di codice aggiunte
	LD	Numero di linee di codice rimosse
	LT	Numero di linee di codice nel file prima della modifica
Purpose Metrics	FIX	Classifica se la modifica è oppure no una correzione di un difetto
History Metrics	NDEV	Numero di sviluppatori che hanno modificato il file
	AGE	Intervallo di tempo medio tra l'ultima e l'attuale modifica
	NUC	Numero di modifiche uniche al file modificato
Experience Metrics	EXP	Esperienza dello sviluppatore
	REXP	Esperienza recente dello sviluppatore
	SEXP	Esperienza dello sviluppatore sul sottosistema d'interesse

Una volta estratte le metriche dal commit, esse possono essere usate come strumento di apprendimento per la costruzione di un modello predittivo. Ovviamente affinché il modello sia il più preciso possibile occorre avere a disposizione una grande quantità di commit ognuno con le proprie metriche.

I dati usati per costruire il modello vengono chiamati *Training data*, dopo che il modello è stato appreso occorre testarlo con dei commit futuri per verificare che effettuino la predizione correttamente, questi dati, che vengono organizzati e formattati in maniera identica a quelli usati per l'apprendimento, vengono chiamati *Testing data*.

In definitiva, la tecnica del Just-in-Time Prediction prevede una serie di passaggi ben

precisi, come mostrato in Figura 6.

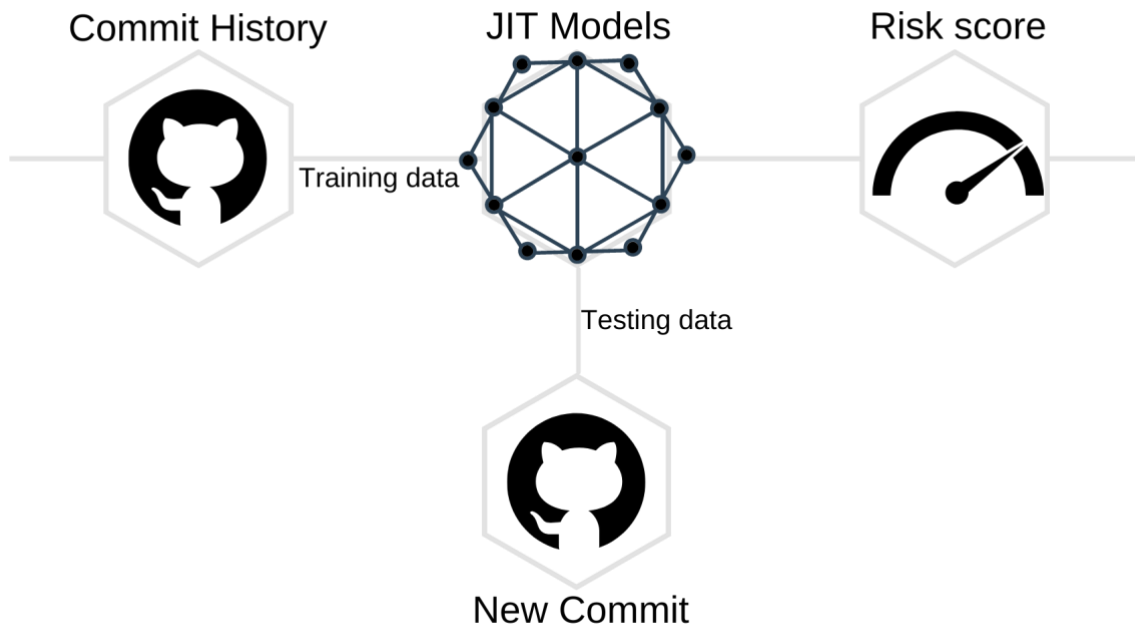


Figura 6: Processo del Just in Time Prediction

3.2 Processo di costruzione e validazione del modello predittivo

Un primo problema da affrontare nella costruzione di un modello predittivo con la tecnica del Just-in-Time Prediction è quello di scegliere quali commit usare per fare il *training* e quali usare per il *testing*, e soprattutto, come usarli senza perdere precisione nei risultati. Per risolvere questo problema possiamo utilizzare due strategie diverse [4], la prima è la seguente:

Time Sensitive Change Classification

Spesso capita di usare commit futuri per prevedere i commit passati, tuttavia, l'utilizzo di una tale strategia porta ad ottenere risultati “artificialmente” buoni in termini di precisione, ma che risultano sbagliati concettualmente nonché temporalmente. Infatti, durante l'estrazione delle metriche e la classificazione dei commit del *training set*, si etichettano come difettosi o non i commit utilizzando tutte le loro informazioni, a volte anche quelle future.

Ecco che sorge, dunque, l'esigenza di portare in conto anche il concetto di tempo nella classificazione dei commit, motivo per cui questa tecnica prende il nome di **Time**

Sensitive Change Classification, essa utilizza solo informazione sui commit passati per etichettare i commit passati stessi, al fine di creare modelli predittivi per i commit futuri, totalmente indipendenti da informazioni future.

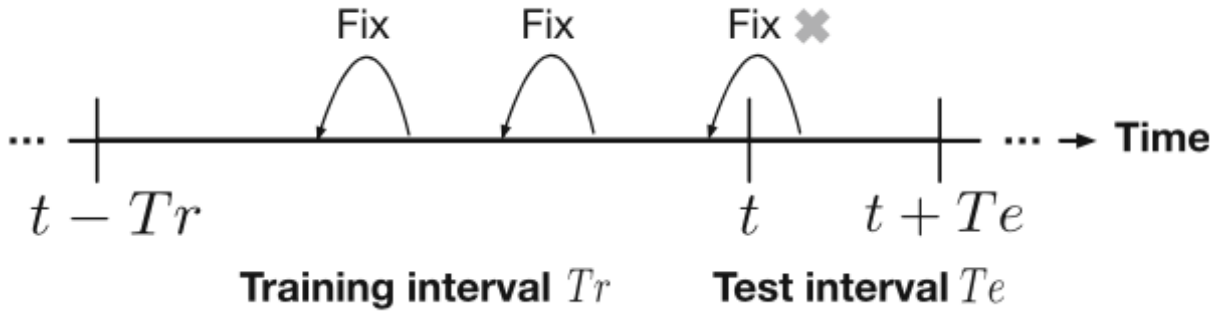


Figura 7: Un esempio di Time Sensitive Change Classification

La Figura 7 mostra un esempio di Time Sensitive Change Classification, in cui il tempo corrente è rappresentato dalla t , l'intervallo di tempo dedicato al *training* è rappresentato con Tr , infine, l'intervallo di tempo dedicato al *testing* è rappresentato con Te . In questo esempio, quindi, usiamo i commit passati per effettuare il training di un modello predittivo per poi effettuare il testing con i commit futuri.

Tuttavia, può capitare che i commit usati nel training siano erroneamente etichettati come non difettosi quando, invece, lo sono perché il tempo impiegato per individuare dei problemi nei commit può essere più lungo di Tr . Un ulteriore grosso problema di questo tipo di classificazione sorge se si prende un intervallo di training Tr molto distante dall'intervallo di testing Te , infatti, in questo lasso di tempo le funzionalità, gli sviluppatori e gli stili di programmazione potrebbero essere cambiati sostanzialmente portando ad avere un modello non coerente con i *testing data*. Per rimediare ai problemi del Time Sensitive Change Classification, si usa come tecnica di classificazione la seguente:

Online Change Classification

Questa strategia di classificazione affronta i problemi del Time Sensitive Change Classification, utilizzando alcuni parametri aggiuntivi rispetto a quest'ultima. In primo luogo, utilizza un *gap* tra l'intervallo di training e l'intervallo di testing, come si può notare dalla Figura 8. Il gap viene utilizzato solo durante l'etichettatura dei commit per consentire

di avere più tempo per rilevare i commit difettosi nell'intervallo di training e rendere più preciso il risultato dell'etichettatura. Questo valore dipende dal tempo impiegato mediamente per correggere un commit difettoso, e quindi, cambia in maniera dinamica in base al programma software analizzato.

Per risolvere il problema legato alla distanza tra intervallo di training ed intervallo di testing, l'operazione di Time Sensitive Change Classification dei commit viene eseguita più volte durante l'aggiornamento dell'intervallo di training, quindi ad ogni iterazione *l'intervallo di training, il gap e l'intervallo di testing* vengono spostati in avanti nel futuro di un certo lasso di tempo.

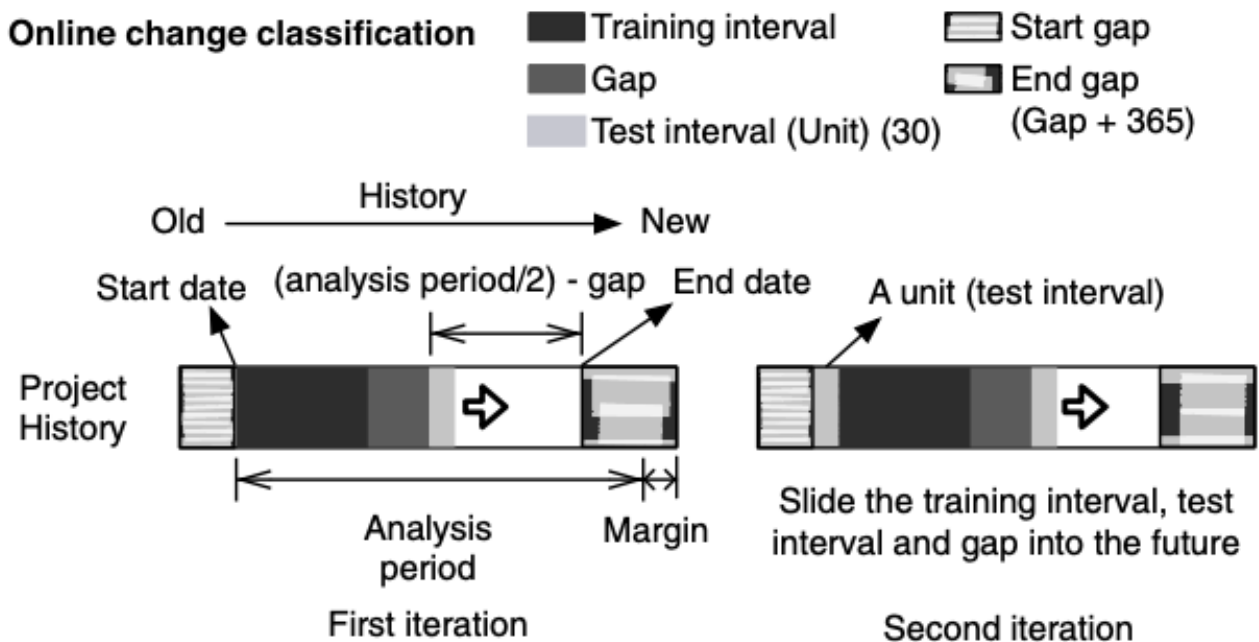


Figura 8: Panoramica dell'Online Change Classification

Questo certo lasso di tempo è chiamato *unit*. Sia l'*unit* che l'intervallo di testing sono parametri che impattano in maniera minore sul risultato degli esperimenti, e per convenienza vengono posti a 30 giorni in entrambi i casi.

Vengono introdotti anche altri parametri come lo *start gap* e l'*end gap* che sono intervalli di tempo che non vengono usati come intervallo di training e intervallo di testing, bensì vengono ignorati perché rappresentano dei dati di non particolare interesse e/o incidenza. Infatti, all'inizio di un progetto software i commit generati sono spesso incoerenti e/o

instabili, mentre, alla fine di un progetto software i commit generati sono spesso etichettati come non difettosi, perché non sono stati ancora rilevati eventuali difetti.

Per definire i valori dei parametri, occorre iniziare con la scelta di una data di inizio (*start date*) che viene spesso presa in coincidenza di una situazione in cui i commit divengono regolari, e quindi dopo la fase transitoria iniziale. Lo *start gap* è l'intervallo tra la data del primo commit e la data di inizio, scelta in precedenza.

Per decidere l'*end gap*, dobbiamo calcolare l'*analysis period*, l'*iteration step size* e l'intervallo di training *Tr*. Questi tre parametri vengono calcolati nel modo seguente:

$$analysis\ period = (CommDate_{latest} - start\ date) - margin,$$

$$iteration\ step\ size = (analysis\ period/2 - gap)/unit,$$

$$Tr = iteration\ step\ size \cdot unit,$$

dove:

- *CommDate_{latest}* è la data dell'ultimo commit
- *margin* è un margine che tiene in considerazione eventuali commit difettosi non ancora rilevati
- *Tr* è l'intervallo di training

Attraverso il calcolo dell'*analysis period* si rimuovono i commit difettosi che non vengono rilevati durante l'intervallo di training *Tr*. Invece, per calcolare l'*iteration step size* usiamo il *gap*, il che ci permette di evitare i commit che si trovano negli ultimi giorni di margine e ci assicura di considerare un numero di commit sufficienti per costruire un modello predittivo capace di classificare i commit nell'intervallo di testing *Te*. Infine, l'intervallo di training *Tr* viene deciso in base all'*iteration step size* ed in base all'*unit*. L'*end date* e l'*end gap* vengono, invece, calcolati come segue:

$$end\ date = start\ date + (Tr + gap + (iteration\ step\ size \cdot unit)),$$

$$end\ gap = CommDate_{latest} - end\ date.$$

Tuttavia, spesso l'*end gap* viene scelto con un margine di 365 sul *gap*, ciò vuol dire che per il suo calcolo basta prendere il *gap* e sommare a quest'ultimo proprio 365, senza usare la formula precedente.

3.2.1 Estrazione delle metriche tramite Commit Guru

Per l'estrazione delle change metrics illustrate nella Tabella 1, useremo il tool Commit Guru [5], il cui scopo è quello di analizzare i commit di un repository GitHub al fine di individuare i commit che hanno introdotto bug. La Homepage di Commit Guru si presenta come in Figura 9, ci viene fornita subito la possibilità di analizzare un nuovo repository (1) inserendo il suo URL, ci viene anche data la possibilità di analizzare un particolare branch (cfr. [paragrafo 2.1.1](#)) del repository.

Repository	Total Commits	Risky Commits (%)	Analysis Time
jackrabbit(trunk)	8,900	20%	Analyzed 4 minutes ago

Figura 9: Homepage di Commit Guru

Commit Guru è composto da due componenti che lavorano a diversi livelli di astrazione, la prima componente è il back-end, il cui codice è disponibile all'indirizzo https://github.com/CommitAnalyzingService/CAS_CodeRepoAnalyzer, il compito di questa componente è quello di analizzare i commit ed estrarre da essi le metriche.

La seconda componente è il front-end, anch'esso disponibile gratuitamente all'indirizzo https://github.com/CommitAnalyzingService/CAS_Web, la quale è responsabile della visualizzazione dei dati analizzati fornendo statistiche e previsioni sui commit difettosi.

Una volta fornita una panoramica generale sul funzionamento di Commit Guru, focalizziamoci adesso su come visualizzare ed interpretare i dati che il tool ci mette a disposizione. Innanzitutto, andiamo nella pagina in cui sono contenuti tutti i repositories analizzati da Commit Guru, Figura 10 (2), all'interno di questa pagina ci vengono fornite subito alcune informazioni riguardo i repositories. Ad esempio, in Figura 10 (3) possiamo vedere che del repository “jackrabbit” è stato analizzato il branch “trunk” al cui interno sono stati effettuati 8900 commit dei quali è stato riscontrato che il 20% sono rischiosi perché introducono dei bug.

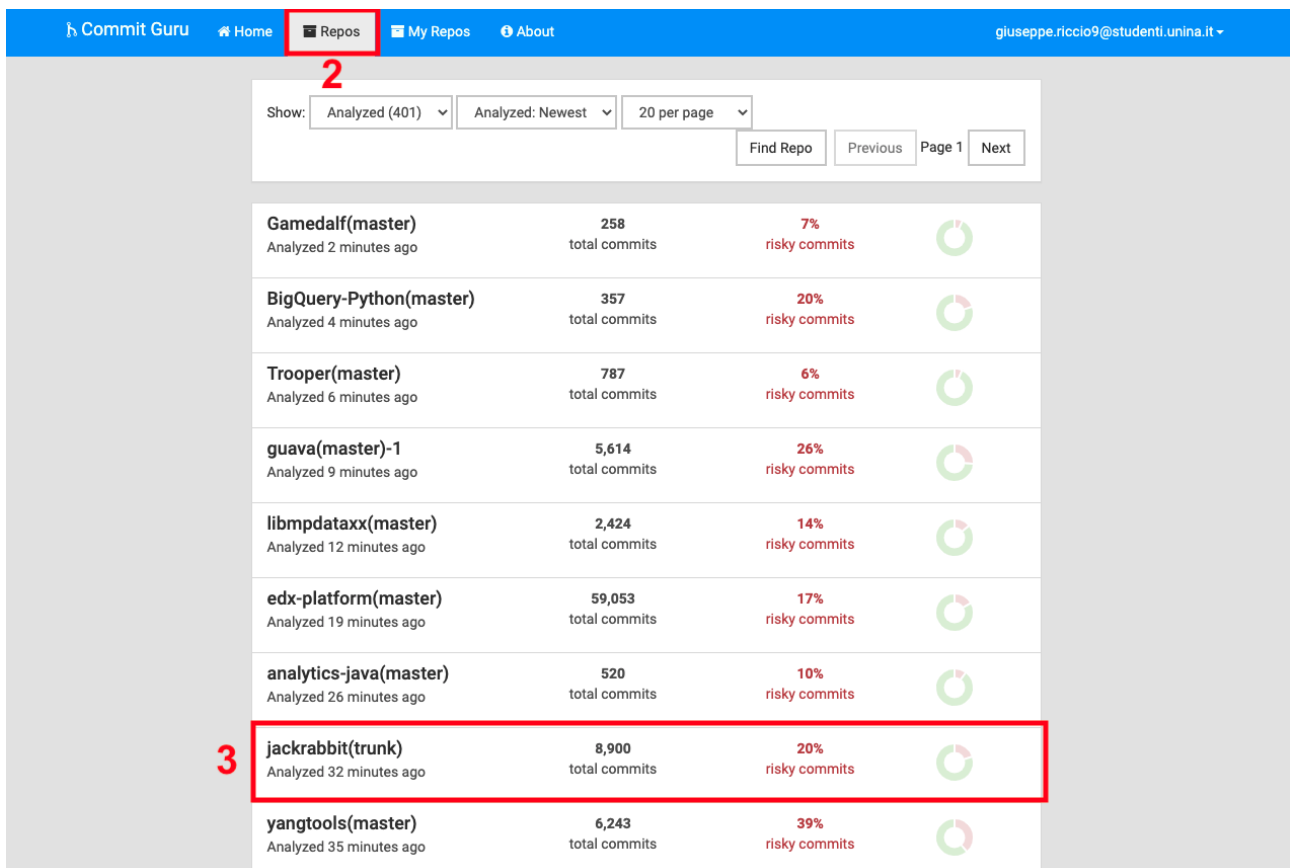


Figura 10: Repositories analizzate da Commit Guru

Cliccando sul repository è possibile ottenere maggiori informazioni, in particolare nella Figura 11 si può notare come vengano riportati tutti i valori in media delle metriche estratte dai commit, sia nel caso appartengano a commit che introducono bug (colorati in rosso) che nel caso appartengano a commit che non introducono bug (colorati in verde). Inoltre, sempre dalla Figura 11, si può notare come ci siano alcune metriche contrassegnate con l'asterisco (*), quest'ultime sono le metriche che statisticamente sono rilevanti nella classificazione di un commit come difettoso o meno, quindi, nell'esempio in questione per costruire il modello predittivo potrebbero essere prese in considerazione solo le 11 metriche con l'asterisco.

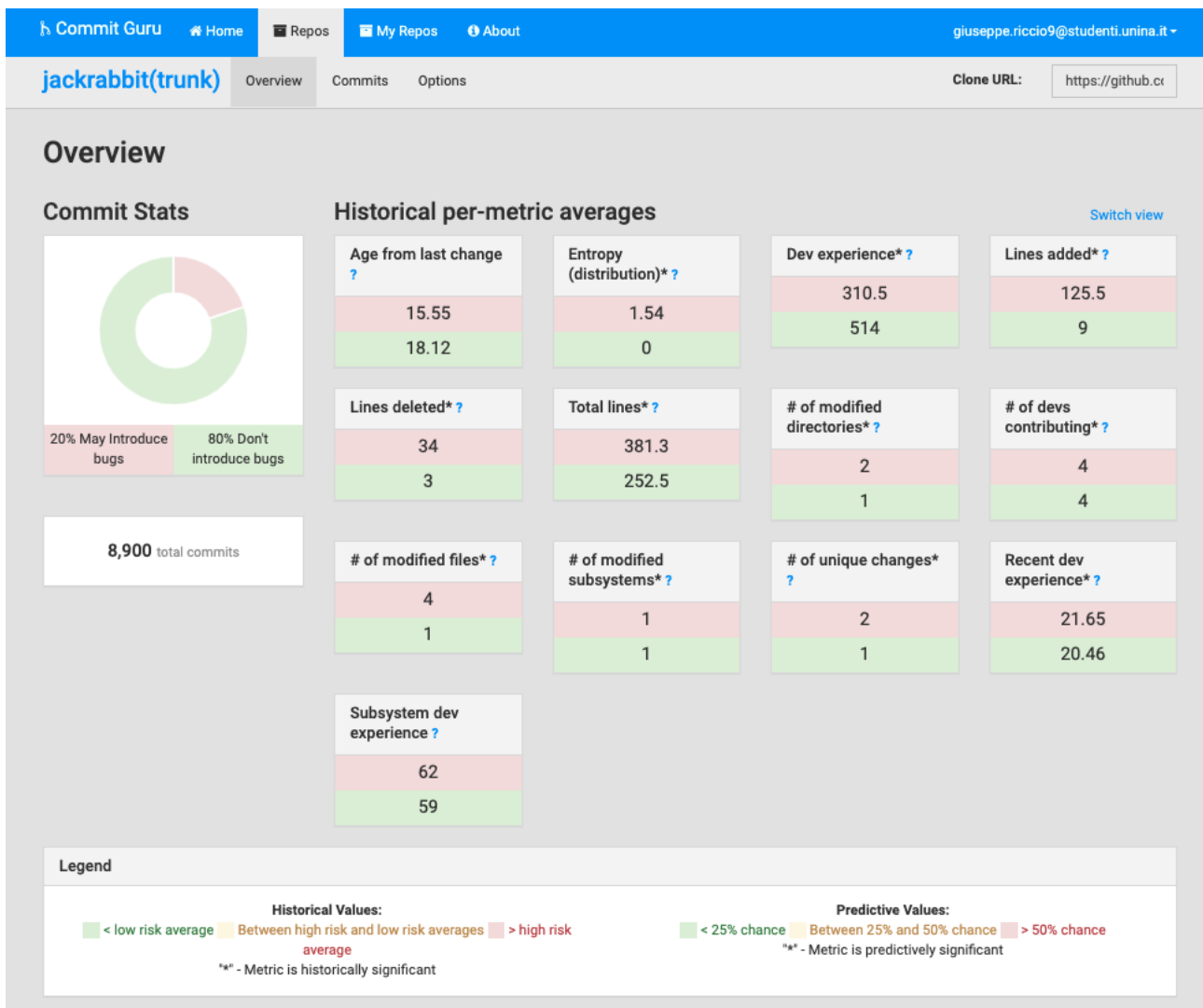


Figura 11: Metriche estratte da Commit Guru

Spostandoci nella sezione *Commits*, come si evince dalla Figura 12 (4), possiamo andare a vedere nel dettaglio i valori delle metriche per ogni singolo commit, nonché vedere la classificazione (5) assegnata a ciascun commit sulla base delle analisi effettuate da Commit Guru.

The screenshot displays the Commit Guru interface. At the top, there's a navigation bar with links to Home, Repos, My Repos, and About. The 'jackrabbit(trunk)' repository is selected, and the 'Commits' tab is active, indicated by a red box and the number 4. Below the navigation bar, the 'Commits' section is shown with various filters and sorting options. A dropdown menu for classification is open, showing a list of categories: Classification (checked), Corrective, Feature addition, Non-functional, Merge, Perfective, and Preventative. This dropdown is highlighted by a red box and the number 5. The main area lists several commits with their hashes, messages, and authors.

Commit Hash	Commit Message	Author	Date
9ceb432501	JCR-4721: Update Logback version to 1.2	Julian Reschke	August 19th 2021, 2:00 pm
95386e1092	JCR-4715: Update slf4j dependency to 1.7.32	Julian Reschke	August 19th 2021, 1:26 pm
7ec32abb00	JCR-4720: webapp: update htmlunit dependency to 2.40.0	Julian Reschke	August 19th 2021, 12:49 pm
99c85e305a	JCR-4719: it-osi: update felix.framework to 7.0.1 git-svn-id: https://svn.apache.org/repos/asf/jackrabbit/trunk	Julian Reschke	August 18th 2021, 10:09 pm
ab34e6edf2	JCR-4718: Upgrade Commons VFS to 2.9.0 git-svn-id: https://svn.apache.org/repos/asf/jackrabbit/trunk	Julian Reschke	August 18th 2021, 9:23 pm
3825a599d4	JCR-4717: Update commons-io dependency to 2.11.0 git-svn-id: https://svn.apache.org/repos/asf/jackrabbit/trunk	Julian Reschke	August 18th 2021, 8:12 pm
cd652325aa	JCR-4716: Update tomcat dependency to 8.5.70 git-svn-id: https://svn.apache.org/repos/asf/jackrabbit/trunk	Julian Reschke	August 18th 2021, 4:30 pm
3985e79f45	JCR-4715: Update slf4j dependency to 1.7.32 git-svn-id: https://svn.apache.org/repos/asf/jackrabbit/trunk	Julian Reschke	August 18th 2021, 2:57 pm
d48ca6b85f	JCR-4712: Update Tika dependency to 2.0.0 git-svn-id: https://svn.apache.org/repos/asf/jackrabbit/trunk	Julian Reschke	August 18th 2021, 5:22 am
35d5732bc1	JCR-4713: avoid use of deprecated tika IOExceptionWithCause ...	Julian Reschke	August 10th 2021, 4:39 pm
6ed753e4dc	JCR-4704: Update Tika dependency to 1.27 git-svn-id: https://svn.apache.org/repos/asf/jackrabbit/trunk	Julian Reschke	July 9th 2021, 2:55 pm
60b688841b	[maven-release-plugin] prepare for next development iteration	Julian Reschke	July 6th 2021, 5:25 am
ebfd5a6212	[maven-release-plugin] prepare release jackrabbit-2.21.7 git-svn-id: https://svn.apache.org/repos/asf/jackrabbit/trunk	Julian Reschke	July 6th 2021, 5:25 am
a2c08f49de	JCR-4703: Release Jackrabbit 2.21.7 - Candidate Release Note...	Julian Reschke	July 5th 2021, 11:04 am

Figura 12: Analisi e classificazione dei singoli commit

In particolare, ogni commit può appartenere ad una tra le sei classi che il tool riesce a rilevare, questo tipo di classificazione si basa sulla ricerca all'interno del *commit message*, ovvero della descrizione data dagli sviluppatori al commit, di alcune parole chiavi che permettono di associarlo ad una specifica classe. Questa funzione di classificazione può essere riassunta tramite la Tabella 2, in cui è riportata per ogni classe a cui può appartenere un commit le parole chiave ad essa associate:

Tabella 2: Parole chiave usate da Commit Guru per classificare un commit

Classe	Parole chiave	Spiegazione
Corrective	bug, fix, wrong, error, fail, problem, patch	Correzione di bug rilevati
Feature Addition	new, add, requirement, initial, create	Aggiunta di nuove funzionalità al software
Merge	merge	Fusione con altre parti del sistema
Non Functional	doc	Aggiunta di documentazione senza toccare il codice
Perfective	clean, better	Ottimizzazione di una parte e/o funzionalità del codice
Preventive	test, junit, coverage, asset	Azioni di test e prevenzione da eventuali malfunzionamenti

Infine, è possibile scaricare tutti i commit con le relative metriche in un file formato .csv, tramite la sezione *Options*, come illustrato in Figura 13 (6), questo file prevede i seguenti 31 attributi: *commit_hash*, *author_name*, *author_date_unix_timestamp*, *author_email*, *author_date*, *commit_message*, *fix*, *classification*, *linked*, *contains_bug*, *fixes*, *ns*, *nd*, *nf*, *entropy*, *la*, *ld*, *fileschanged*, *lt*, *ndev*, *age*, *nuc*, *exp*, *rexp*, *sexp*, *glm_probability*, *rf_probability*, *repository_id*, *issue_id*, *issue_date* e *issue_type*.

Figura 13: Download del file con le metriche in formato .csv

3.2.2 Costruzione del modello con Weka

Come modelli predittivi dei commit difettosi useremo il modello di Logistic Regression (LR) e il modello di Random Forest (RF). I concetti che stanno alla base di questi due modelli sono le seguenti:

- **Logistic Regression (LR):** LR è un modello predittivo molto usato nell'ambito della predizione di commit difettosi, in quanto funziona molto bene quando si trattano variabili di tipo dicotomiche (ovvero, che assumono solo due valori) e/o continue. In questo caso si costruisce un modello lineare che tiene conto di tutte le metriche come variabili di interesse. LR pone l'output di questo modello lineare in ingresso ad una funzione sigmoide (*logistica*). L'output della funzione sigmoide corrisponde alla probabilità che un certo commit sia difettoso o meno.
- **Random Forest (RF):** RF è un modello di apprendimento integrato, ovvero costituito da un complesso di diversi modelli. RF, infatti, costruisce vari alberi decisionali basati su dei sottoinsiemi delle metriche. Infine, RF unisce tutti i risultati degli alberi decisionali e fornisce la probabilità che il commit sia difettoso o meno. Questo modello essendo basato sugli alberi decisionali, dipende fortemente dagli attributi, in questo caso le metriche, che si adoperano e cerca, dunque, di ottimizzare questi parametri. Ad esempio, una metrica viene preferita

alle altre se presenta un grado di entropia minore, perché ciò vuol dire che è in grado di ottenere dei risultati più precisi.

Nel nostro caso di studio useremo un software di machine learning che racchiude una collezione di algoritmi che permettono di manipolare dati, effettuare classificazioni, regression test e visualizzazione dei dati, tra questi algoritmi troviamo anche il Logistic Regression ed il Random Forest. Il software in questione è Weka, scritto in Java, è un tool open source [6] sviluppato da un gruppo di ricerca della Waikato University in Nuova Zelanda. Il programma all'avvio si presenta come in Figura 14, la sezione di interesse è quella di “*Explorer*”, che vedremo nel dettaglio nel prossimo capitolo.

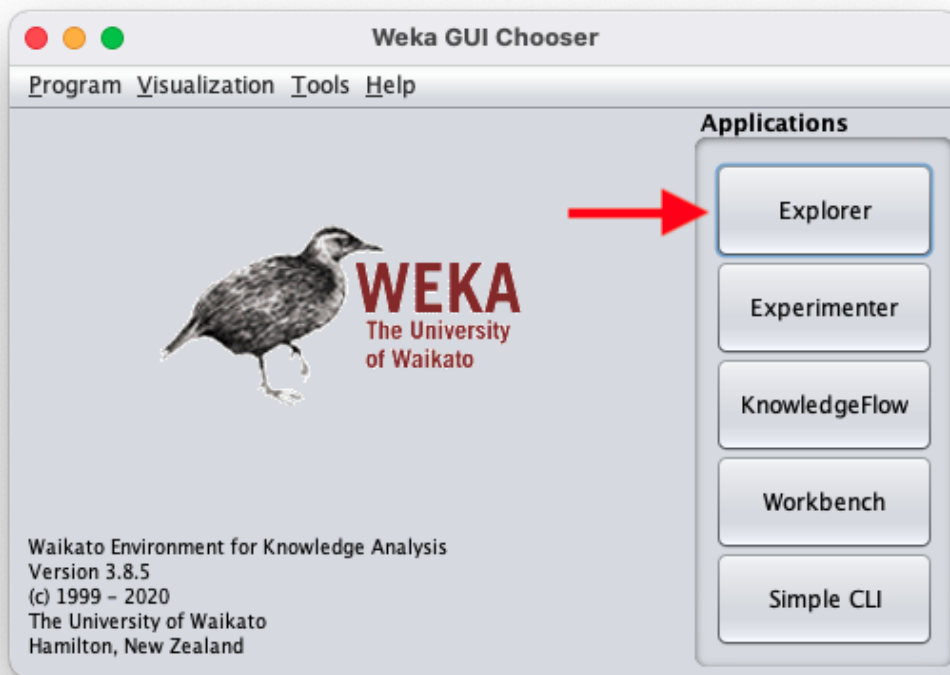


Figura 14: Menù del software Weka

Per costruire i modelli che abbiamo illustrato precedentemente, all'interno della finestra che si è aperta dopo aver cliccato su “*Explorer*” dobbiamo andare nella sezione “*Classify*” così come mostrato in Figura 15. A questo punto dobbiamo scegliere all'interno dei vari *classifier* messi a disposizione da Weka, uno dei due modelli che ci interessano ovvero, Logistic o Random Forest, il tool a questo punto costruirà il modello

scelto fornendo anche alcune informazioni essenziali, che possono essere utili ai fini della valutazione della bontà del modello stesso.

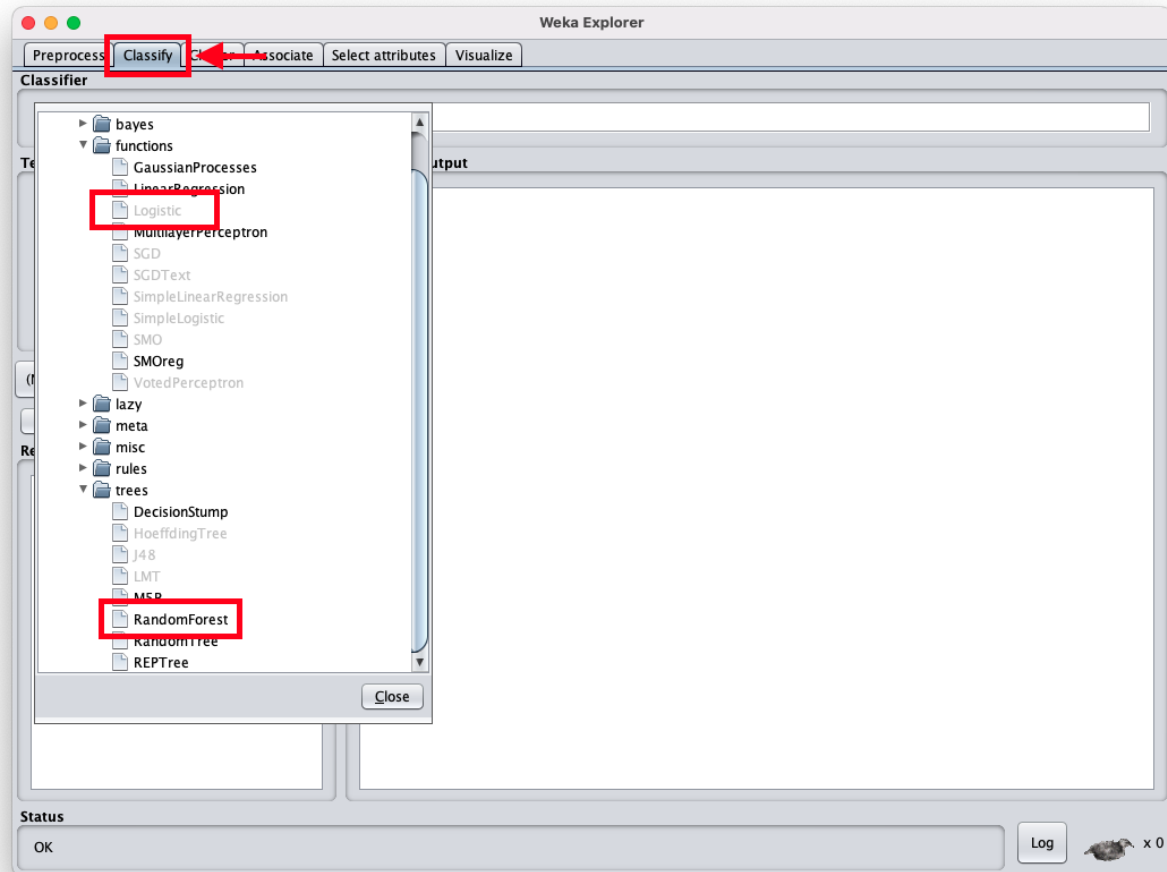


Figura 15: Scelta del modello su Weka

3.2.3 Misurazione delle performance delle metriche

Per misurare l'impatto delle metriche sulla previsione dei commit difettosi, utilizziamo tre misure di valutazione [4]: l'area sotto la curva caratteristica di funzionamento del ricevitore (Area Under the receiver operating characteristic Curve, AUC), il coefficiente di correlazione di Matthews (Matthews Correlation Coefficient, MCC) ed il punteggio Brier (Brier score).

La curva ROC viene disegnata tracciando il valore del True Positive Rate (TPR), anche detto *sensibilità*, rispetto al valore del False Positive Rate (FPR), anche detto *fall-out* (può essere calcolato come $1 - \text{specificità}$). Questi valori per essere calcolati hanno bisogno di conoscere le distribuzioni di probabilità delle predizioni ottenute, in particolare avremo

che:

- se il valore predetto è positivo p' ed il valore vero è anch'esso positivo p , viene chiamato vero positivo (*true positive* - TP);
- se il valore predetto è positivo p' ed il valore vero è negativo n , viene chiamato falso positivo (*false positive* - FP);
- se il valore predetto è negativo n' ed il valore vero è negativo n , viene chiamato vero negativo (*true negative* - TN);
- se il valore predetto è negativo n' ed il valore vero è positivo p , viene chiamato falso negativo (*false negative* - FN).

A questo punto abbiamo tutto quello che serve per calcolare il TPR ed il FPR, infatti, questi ultimi possono essere calcolati come segue:

$$TPR = TP/P = TP/(TP + FN)$$

$$FPR = FP/N = FP/(FP + TN)$$

Una volta calcolati questi valori è facile disegnare la curva ROC, da cui può essere calcolata l'area sottesa, in Figura 16 vengono rappresentate alcune curve ROC tipiche di cui è già stato calcolato il valore di AUC (che è compreso sempre tra 0 e 1).

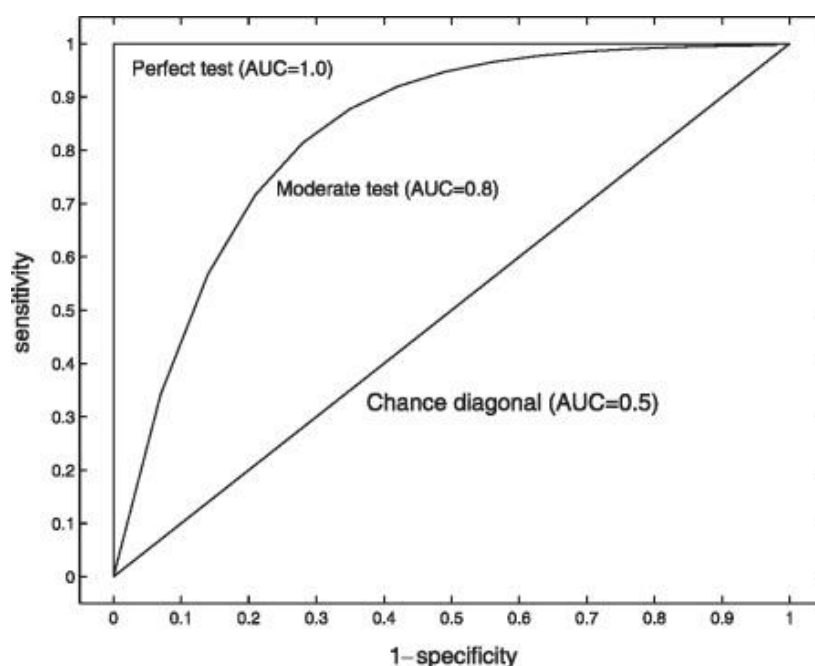


Figura 16: Valori di AUC per particolari curve ROC

Il coefficiente di correlazione di Matthews è molto usato nell'ambito del machine learning per misurare la qualità di classificazioni binarie. Per il suo calcolo vengono presi in considerazione come nel caso del AUC, i valori di veri e falsi positivi/negativi, tuttavia, in questo caso il valore del MCC varia tra -1 ed 1. Il MCC è 1 quando la predizione è perfetta ovvero i valori predetti corrispondono ai valori veri, il valore è 0 quando non si può dire nulla sulla predizione che è del tutto incerta, infine, il valore è -1 quando i valori predetti sono diversi dai valori veri. La formula per il calcolo del MCC è la seguente:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Equazione 1: Calcolo del MCC

Il punteggio Brier è una funzione che misura l'accuratezza delle previsioni probabilistiche, esso in caso di stime unidimensionali, è strettamente equivalente all'errore quadratico medio applicato alle probabilità previste.

Il punteggio Brier è applicabile alle classificazioni di natura binaria o categorica, di cui è possibile cioè effettuare una previsione del valore con una certa probabilità. Più precisamente, per tutti gli elementi in un insieme di N previsioni, il punteggio Brier misura la differenza quadratica media tra:

- Il valore predetto con una certa probabilità f_t
- Il valore effettivo o_t

Traducendo questo in formule, avremo che:

$$BS = \frac{1}{N} \sum_{t=1}^N (f_t - o_t)^2$$

Equazione 2: Calcolo del Brier score

Pertanto, più basso è il punteggio Brier per una serie di previsioni, migliori sono le previsioni effettuate perché ciò vuol dire che il valore predetto si discosta di poco rispetto al valore effettivo. Si noti che il punteggio di Brier, sempre nell'ipotesi di classificazioni binarie, assume un valore compreso tra 0 ed 1, poiché questo è il quadrato della più grande differenza possibile tra una probabilità prevista (che deve essere tra 0 ed 1) e il risultato effettivo (che può assumere solo valori di 0 o 1).

3.2.4 Validazione del modello tramite testing data

Per validare il modello costruito da Weka, sulla base dei *training data* dei commit passati, si possono utilizzare diverse tecniche, la prima è quella di usare una *cross-validation*. La *cross-validation* prevede la suddivisione dell'insieme di dati forniti in ingresso in k-parti, chiamate *folds*, il valore di k viene deciso sulla base di analisi e stime sul particolare caso di studio. Le k-parti dell'insieme di dati hanno uguale dimensione e ad ogni passo viene scelta la k^a parte come *training data*, mentre le restanti parti sono i *training data* per costruire il modello.

Una seconda tecnica di validazione del modello, che è anche quella usata in questo elaborato, prevede la suddivisione statica dell'insieme di dati in due parti, la prima è proprio quella dei *training data* che forniscono il supporto per l'apprendimento del modello tramite un insieme di commit, e delle loro metriche, la cui classificazione è già ben nota. La seconda parte dei dati viene usata come *testing data*, ovvero si prendono una certa quantità di commit, la cui classificazione è sconosciuta e si cerca tramite il modello predittivo appreso di effettuare una predizione sulla loro natura.

La percentuale di dati usati per il training e quella usata per il testing viene scelta manualmente, sulla base dei valori dei parametri dell'Online Change Classification calcolati come enunciato nel [paragrafo 3.2](#).

Capitolo 4: Caso di studio: Immuni App, Android vs. iOS

Per l'applicazione delle tecniche di predizione, ed in particolare del Just-in-Time Prediction, prenderemo come caso di studio quello di Immuni, un'app per Android ed iOS, che serve al tracciamento delle persone positive. Inoltre, dall'introduzione del *green pass* l'app di Immuni è diventata anche un metodo per scaricare e salvare in modalità offline il proprio certificato verde rappresentato con un codice QR. In Figura 17, possiamo vedere l'interfaccia grafica dell'applicazione, in cui viene mostrato che il servizio per la notifica delle esposizioni è attivo (1), poi è possibile scaricare e visualizzare il EU Digital Covid Certificate (2) oppure segnalare la propria positività in maniera tale da informare tutti coloro che sono stati a contatto con il soggetto in questione (3).

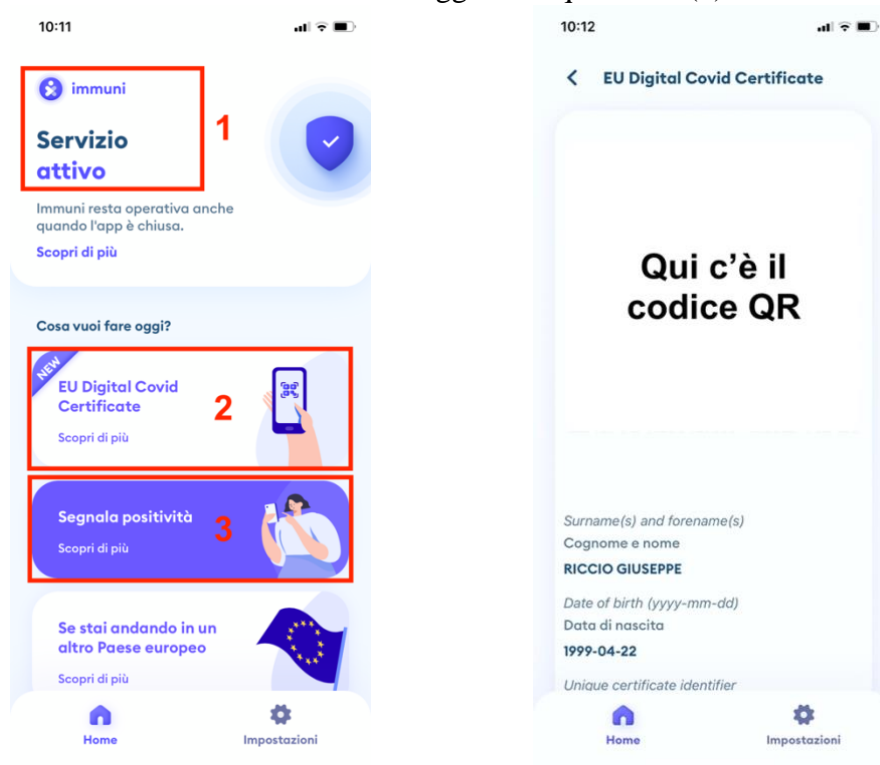


Figura 17: Panoramica app Immuni

4.1 Progettazione del caso di studio

Al fine dell'uso delle tecniche di predizione, ci serviamo dei repositories GitHub di Immuni, infatti i file ed i codici sorgente dell'app sono disponibili in maniera open source ai seguenti link:

- Android: <https://github.com/immuni-app/immuni-app-android>
- iOS: <https://github.com/immuni-app/immuni-app-ios>

Come è possibile notare aprendo i link forniti, il codice per l'app Android è scritto con **Kotlin** [7], un linguaggio di programmazione object-oriented sviluppato dall'azienda di software JetBrains. Questo linguaggio si basa sulla Java Virtual Machine (JVM), la sua sintassi è simile a quella di Java e per questo è molto facile convertire un programma da Java a Kotlin, tuttavia, esso si ispira anche a Go (linguaggio di programmazione di Google). Kotlin ha preso particolarmente piede in ambito Android, in quanto integrato all'interno dell'ambiente di sviluppo Android Studio a partire dalla versione 3.0, ed è addirittura consigliato da Google stessa per lo sviluppo di app Android.

Per quanto riguarda, invece, l'app iOS essa è scritta con **Swift** [8], un linguaggio di programmazione creato da Apple stessa per scrivere in maniera intuitiva app per tutti i suoi dispositivi, dagli iPhone ai MacBook passando per le Apple TV, nonché sviluppare app per Linux. Questo linguaggio è open source e permette la perfetta integrazione con il codice Objective-C, il che permette di sfruttare funzioni già sviluppate integrandole in maniera rapida nell'app, inoltre, questo linguaggio presenta delle prestazioni notevolmente superiori rispetto a linguaggi tradizionali. Ad esempio, come riportato nella pagina ufficiale di Apple, Swift è circa 2,6 volte più veloce di Objective-C ed addirittura 8,4 volte più veloce di Python 2.7 nell'esecuzione di algoritmi di ricerca.

4.1.1 Estrazione delle metriche e definizione dei parametri di classificazione

Come illustrato nel [paragrafo 3.2.1](#), per l'estrazione delle metriche utili alla costruzione di un modello predittivo per l'individuazione dei commit difettosi utilizziamo il tool Commit Guru. Innanzitutto, andiamo a caricare sul sito di Commit Guru, l'URL del repository dove è contenuto il codice dell'app Immuni, questo passaggio viene ripetuto due volte una

volta per la versione Android l'altra per la versione iOS, al termine di questa operazione nella sezione "My Repos" è possibile trovare i due repositories analizzati, come si può notare anche dalla Figura 18 (1).

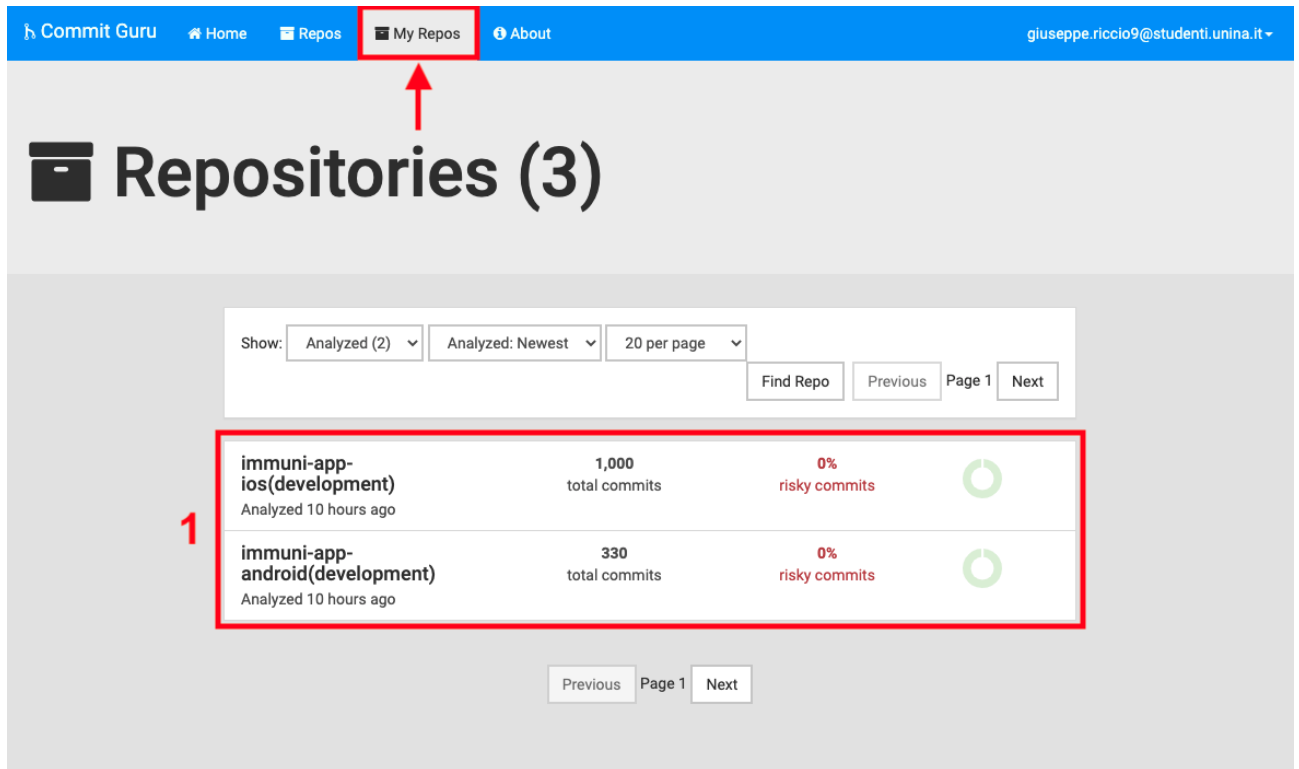


Figura 18: Repositories analizzati da Commit Guru

Cliccando sui due repositories è possibile ottenere i valori medi delle metriche, nonché individuare quali sono le metriche che maggiormente impattano sulla classificazione dei commit come difettosi o meno. Per quanto riguarda la versione iOS di Immuni, come si evince dalla Figura 19, possiamo notare che essa presenta un rischio dello 0% di commit difettosi e nessuna metrica prevale sulle altre nell'impatto sulla classificazione. Invece, nella Figura 20, si può notare come anche la versione Android presenti un rischio dello 0% nella presenza di commit difettosi, tuttavia, in questo caso la metrica che individua il numero di sottosistemi modificati (NS, diffusion metrics) è significativa rispetto alle altre nella predizione.

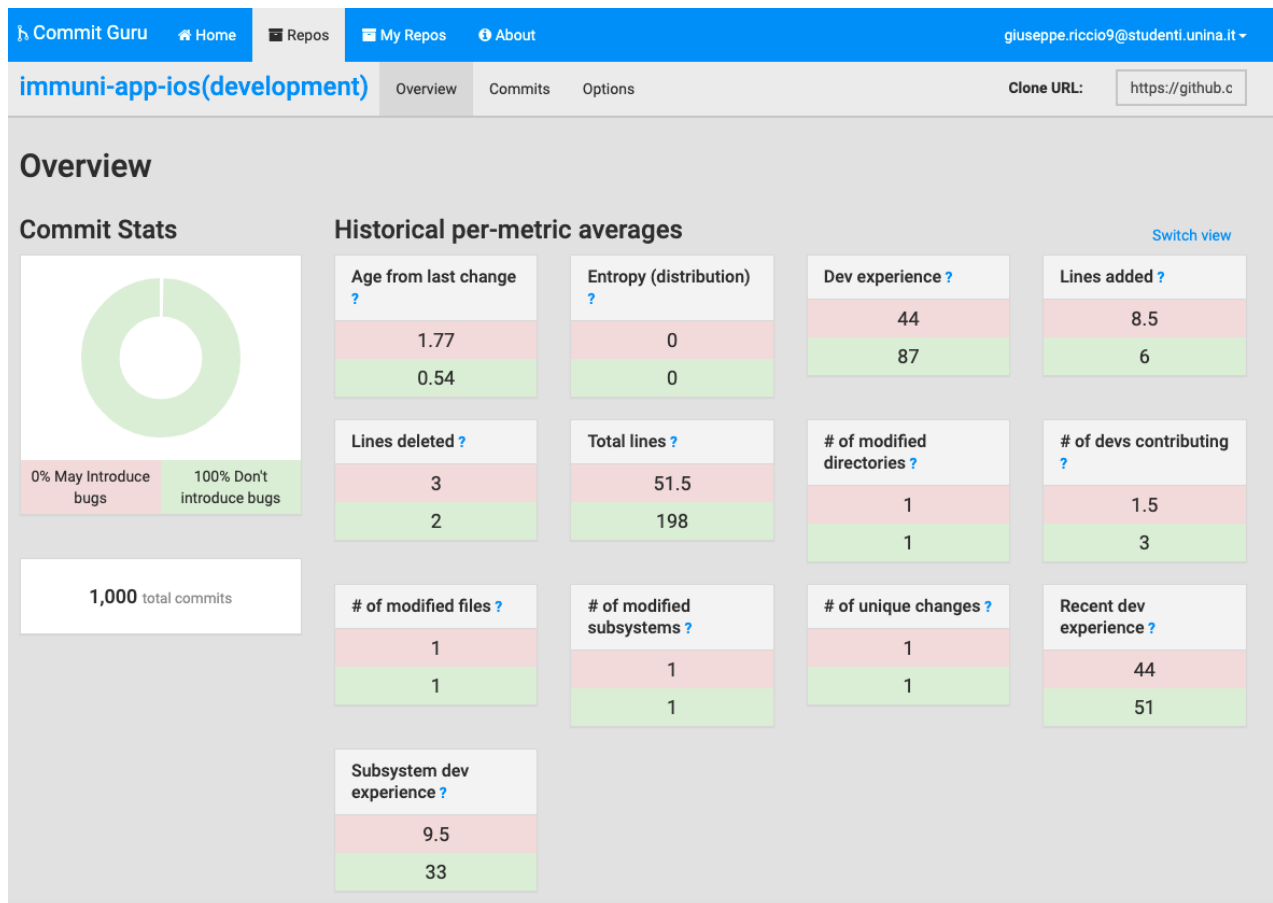


Figura 19: Valori medi delle metriche per Immuni versione iOS

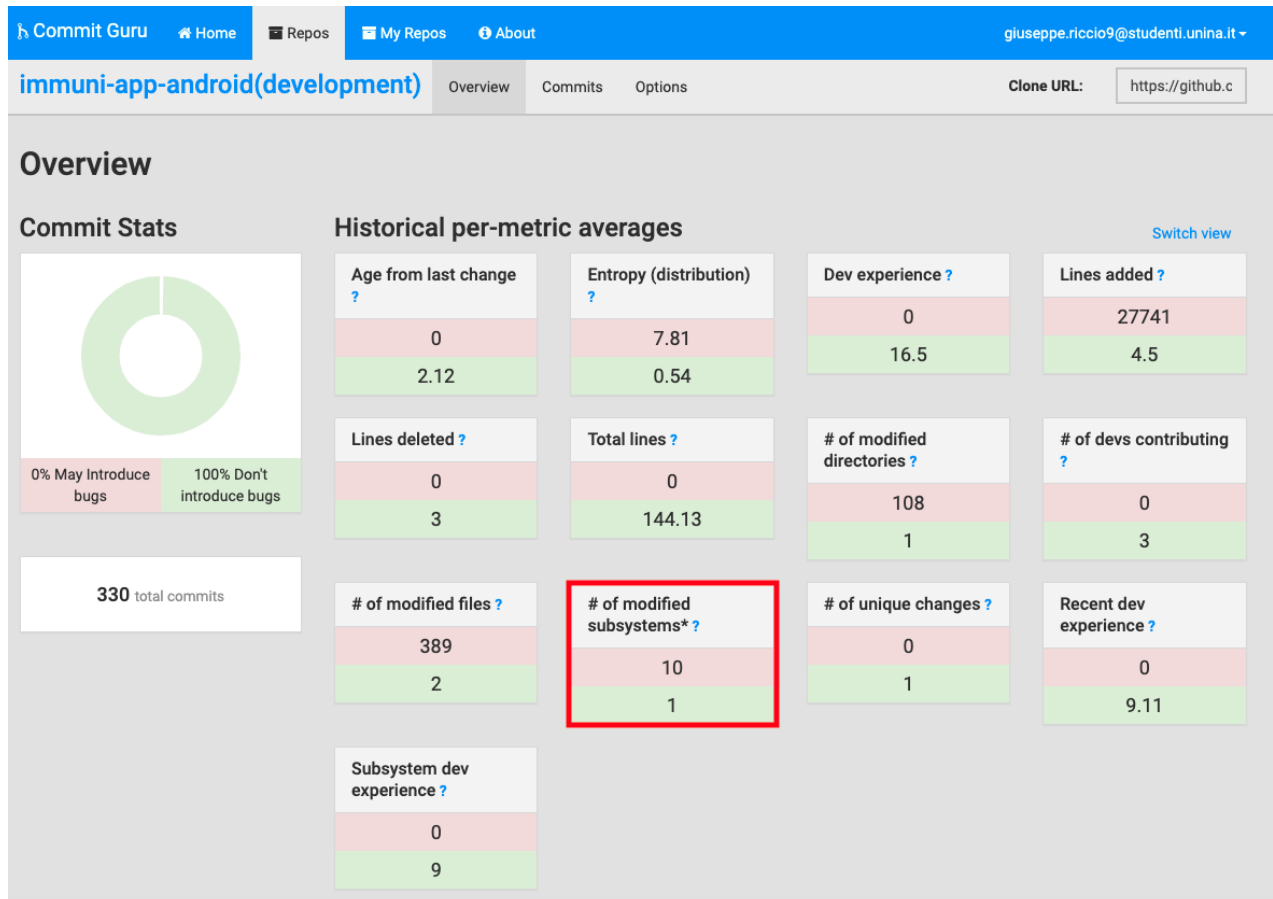


Figura 20: Valori medi delle metriche per Immuni versione Android

Dopo la panoramica sui valori medi delle metriche e l'individuazione di quelle più significative, possiamo a questo punto scaricare i due file in formato .csv, sia per la versione iOS che per quella Android, in cui sono presenti i valori in dettaglio per ogni singolo commit, come mostrato in Figura 21.

ns	nd	nf	entropy	la	ld	lt	fix	ndev	age	nuc	exp	rexp	sexp
0.0	0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0
3.0	3.0	3.0	14.854.752.972.273.300	4.0	6.0	19.166.666.666.666.600	True	7.0	2.677.977.237.654.320	2.0	45.0	4.289.070.319.658.550	19.666.666.666.666.600
0.0	0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0
2.0	2.0	2.0	0.5435644431995964	10.0	6.0	182.5	True	7.0	2.193.240.162.037.030	2.0	44.0	4.701.033.169.187.190	7.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0
2.0	2.0	2.0	0.41381685030363374	17.0	7.0	144.0	True	6.0	2.274.451.388.888.880	2.0	43.0	73.272.591.168.128.600	26.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0
1.0	1.0	2.0	0.8112781244591328	4.0	4.0	301.0	True	2.0	0.0010763888888888889	1.0	42.0	12.018.456.494.772.200	41.0
1.0	3.0	4.0	19.503.749.903.792.700	50.0	24.0	319.5	True	7.0	602.984.375	3.0	41.0	11.018.456.494.772.200	40.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0
1.0	1.0	1.0	0.0	2.0	0.0	186.0	True	6.0	9.924.328.703.703.700	1.0	2.0	0.13392857142857142	1.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0
1.0	7.0	7.0	27.309.641.009.918.600	27.0	61.0	431.0	True	7.0	3.662.627.314.814.810	4.0	40.0	15.371.470.915.278.900	39.0
1.0	2.0	2.0	0.8224042259549891	0.0	35.0	239.0	True	2.0	5.683.310.185.185.180	2.0	39.0	14.876.951.497.075.300	38.0
1.0	1.0	1.0	0.0	1.0	1.0	567.0	True	7.0	4.926.620.370.370.370	1.0	38.0	14.379.583.076.022.700	37.0
1.0	1.0	1.0	0.0	1.0	1.0	502.0	True	6.0	12.732.060.185.185.100	1.0	37.0	1.613.325.954.661.090	36.0
3.0	3.0	3.0	0.9737837298712213	40.0	8.0	13.933.333.333.333.300	True	7.0	14.338.001.543.209.800	3.0	36.0	15.299.926.213.277.600	15.333.333.333.333.300
1.0	9.0	10.0	28.818.665.431.825.200	24.0	24.0	362.1	True	8.0	10.214.479.166.666.600	3.0	2.0	2.0	0.0

Figura 21: Metriche estratte da Commit Guru

A questo punto ci rimane da definire i parametri per la **Online Change Classification** come definiti nel [paragrafo 3.2](#), in particolar modo procediamo nel modo seguente:

Immuni App Android

La data del primo commit è 25 maggio 2020, definiamo questo valore come *CommDateoldest*, come *start date* scegliamo il 1° giugno 2020, in quanto a partire da quella data i commit diventano più costanti e regolari. Quindi, possiamo calcolare lo *start gap* come segue:

$$start\ gap = start\ date - CommDateoldest = 7\ (giorni)$$

Definendo un *margin* di 14 giorni, che ci permette di escludere gli ultimi commit che ancora non sono stati classificati come difettosi o meno, ed osservando la storia dei commit risulta che la data dell'ultimo commit effettuato è il 3 agosto 2021. È possibile, quindi, calcolare anche gli altri seguenti parametri:

$$analysis\ period = (CommDate_{latest} - start\ date) - margin = 428 - 14 = 414$$

$$iteration\ step\ size = (analysis\ period/2 - gap)/unit = (207 - 7)/30 = 6,67$$

Nel calcolo dell'*iteration step size* si è scelto come lasso di tempo (*gap*) impiegato dagli sviluppatori per correggere un bug pari a 7 giorni. Il valore di *iteration step size* lo arrotondiamo per eccesso e lo prendiamo proprio uguale a 7 giorni, il che ci permette di calcolare l'*intervallo di training Tr* in questo modo:

$$Tr = iteration\ step\ size \cdot unit = 7 \cdot 30 = 210$$

Infine, ci rimangono da calcolare solo i valori di *end date* ed *end gap*, il che risulta automatico avendo già calcolato tutti gli altri parametri, infatti, risulta:

$$end\ date = start\ date + (Tr + gap + (iteration\ step\ size \cdot unit)) = 2\ agosto\ 2021$$

$$end\ gap = CommDate_{latest} - end\ date = 1$$

Adesso abbiamo tutti i parametri che ci servono per costruire il modello predittivo per l'app Immuni versione Android, non resta che eseguire lo stesso procedimento di calcolo per trovare i parametri per l'app Immuni versione iOS:

Immuni App iOS

Anche in questo caso la data del primo commit è 25 maggio 2020, definiamo questo valore come *CommDateoldest*, come *start date* scegliamo il 3 giugno 2020, in quanto a partire da quella data viene rilasciata la release stabile dell'applicazione. Quindi, possiamo calcolare lo *start gap* come segue:

$$start\ gap = start\ date - CommDateoldest = 9\ (giorni)$$

Definendo un *margin* di 14 giorni, che ci permette di escludere gli ultimi commit che ancora non sono stati classificati come difettosi o meno, ed osservando la storia dei

commit risulta che la data dell'ultimo commit effettuato è il 2 luglio 2021. È possibile, quindi, calcolare anche gli altri seguenti parametri:

$$analysis\ period = (CommDate_{latest} - start\ date) - margin = 394 - 14 = 380$$

$$iteration\ step\ size = (analysis\ period/2 - gap)/unit = (190 - 10)/30 = 6$$

Nel calcolo dell'*iteration step size* si è scelto come lasso di tempo (*gap*) impiegato dagli sviluppatori per correggere un bug pari a 10 giorni. Il valore di *iteration step size* ci permette di calcolare l'*intervallo di training* Tr in questo modo:

$$Tr = iteration\ step\ size \cdot unit = 6 \cdot 30 = 180$$

Infine, ci rimangono da calcolare solo i valori di *end date* ed *end gap*, il che risulta automatico avendo già calcolato tutti gli altri parametri, infatti, risulta:

$$end\ date = start\ date + (Tr + gap + (iteration\ step\ size \cdot unit)) = 8\ giugno\ 2021$$

$$end\ gap = CommDate_{latest} - end\ date = 24$$

I parametri calcolati precedentemente per le due versioni dell'app Immuni, possono essere riassunti nella Tabella 3.

Tabella 3: Parametri dell'Online Change Classification per le versioni di Immuni (valori espressi in giorni)

Progetto	Start gap	End gap	Gap	Unit	(Intervallo di Testing)	Intervallo di Training	Iteration step size
Immuni App Android	7	1	7	30		210	7
Immuni App iOS	9	24	10	30		180	6

4.1.2 Costruzione del modello a partire dal dataset

[DA AGGIUNGERE]

4.1.3 Validazione del modello

[DA AGGIUNGERE]

4.1.4 Risultati ed osservazioni

[DA AGGIUNGERE]

Conclusioni

[DA AGGIUNGERE]



Bibliografia

- [1] “Manifesto per lo sviluppo agile di software”, <https://agilemanifesto.org/iso/it/principles.html>, Consultato: 3 agosto 2021
- [2] Mojtaba Shahina, Muhammad Ali Babara, Liming Zhub, “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices”, https://www.researchgate.net/publication/315381994_Continuous_Integration_Delivery_and_Deployment_A_Systematic_Review_on_Approaches_Tools_Challenges_and_Practices, Consultato: 5 agosto 2021
- [3] Corrado Lombardi, Giulio Destri, “I processi di sviluppo software: L’evoluzione agile ed il DevOps”, http://www.giuliodestri.it/doc/D07_SviluppoSoftwareAgile.pdf, Consultato: 18 agosto 2021
- [4] Masanari Kondo, Daniel M. German, Osamu Mizuno, Eun-Hye Choi, “The impact of context metrics on just-in-time defect prediction”, <https://link.springer.com/article/10.1007/s10664-019-09736-3>, Consultato: 30 luglio 2021
- [5] Rosen Christoffer, Grawi Ben, Shihab Emad, “Commit Guru: Analytics and Risk Prediction of Software Commits”, http://das.encs.concordia.ca/uploads/2016/01/Rosen_FSE2015Tool.pdf, Consultato: 4 agosto 2021
- [6] Waikato University, “Weka 3: Machine Learning Software in Java”, <https://www.cs.waikato.ac.nz/ml/weka/>, Consultato: 4 agosto 2021
- [7] Wikipedia, “Kotlin (linguaggio di programmazione)”, [https://it.wikipedia.org/wiki/Kotlin_\(linguaggio_di_programmazione\)](https://it.wikipedia.org/wiki/Kotlin_(linguaggio_di_programmazione)), Consultato: 23

agosto 2021

[8] Apple, “Swift. Un linguaggio potente e aperto a tutti per creare fantastiche app.”,

<https://www.apple.com/it/swift/>, Consultato: 23 agosto 2021

[9]