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

Introduzione

All'interno di questa tesi di Laurea verranno trattate le tecniche di predizione di commit difettosi nell'ambito dello sviluppo del software con modello Continuous Integration, a tal fine ci avvarremo del supporto di strumenti per il controllo di versione come GitHub.

In particolare, affronteremo attraverso l'uso della metodologia del 'Just-in-Time Prediction', il problema dell'estrazione delle metriche a partire da un log di commit e la conseguente creazione di un file .csv. A partire dalle metriche estratte, che rappresenteranno il cosiddetto *training set*, e con l'uso di particolari algoritmi di *Machine Learning*, verrà costruito un modello predittivo in grado di etichettare ogni commit come difettoso o meno.

Poi ci preoccuperemo di validare questo modello sottoponendogli un insieme di commit, detto *testing set*, di cui verificheremo le predizioni date dal modello al fine di valutare la sua precisione e correttezza nei risultati forniti.

A questo proposito, tratteremo un caso di studio particolare, quello dall'app Immuni nelle due versioni per sistemi operativi mobile, ovvero Android e iOS, al fine di effettuare delle osservazioni sull'efficacia e la precisione di un certo modello predittivo piuttosto che un altro, nonché mettere in pratica il processo di predizione 'Just-in-Time'.

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 operazionale (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.

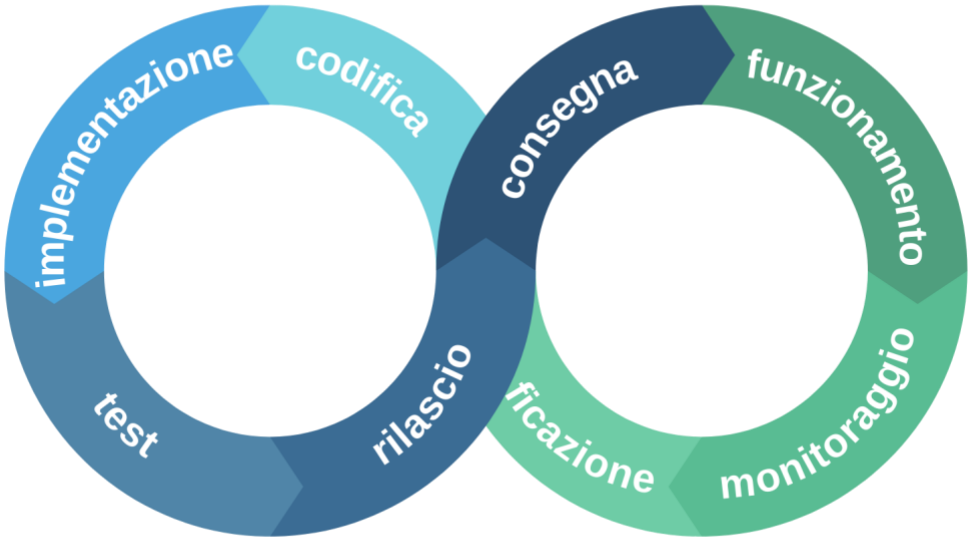


Figura 1: Ciclo DevOps

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 operazionale, 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".

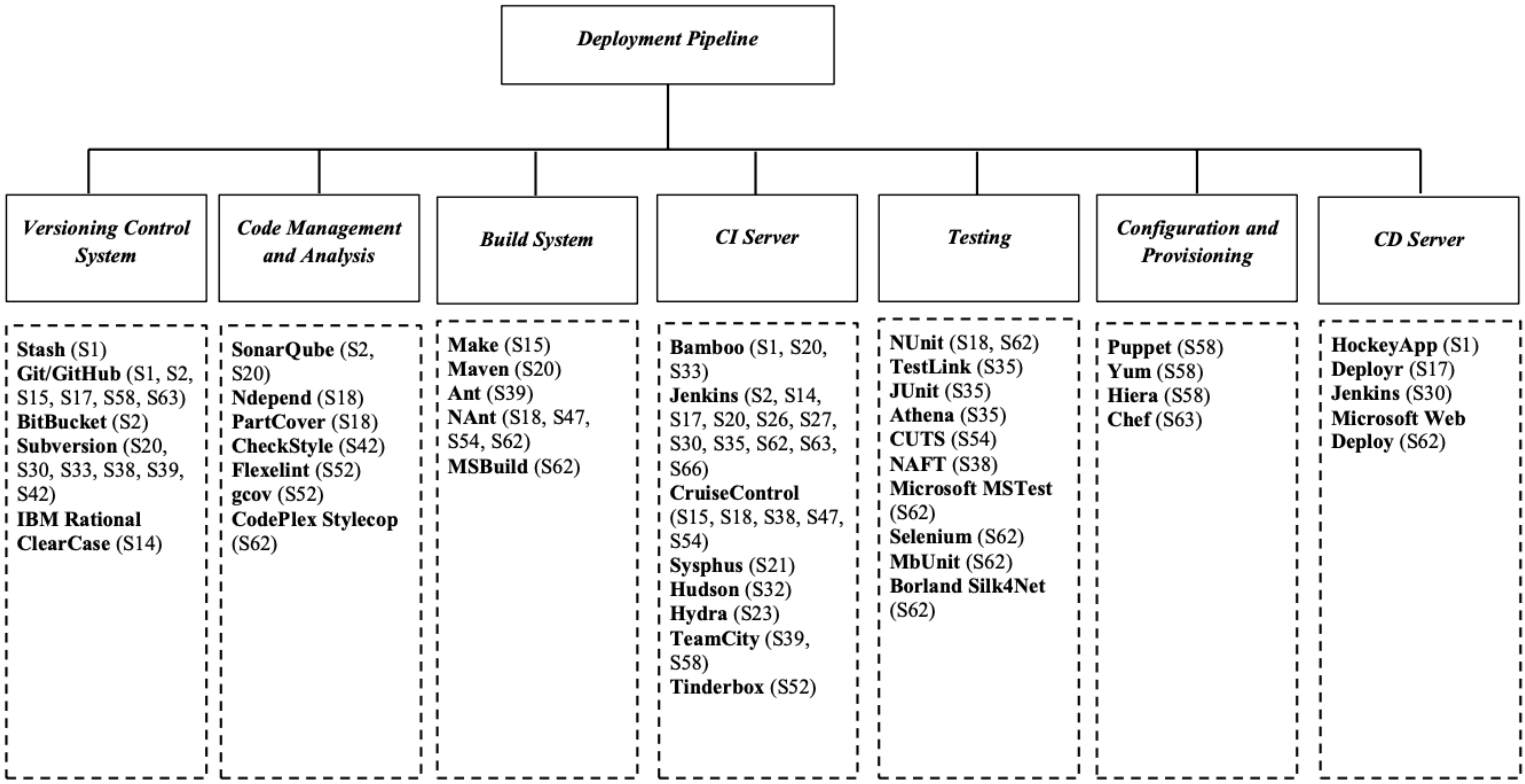


Figura 3: Strumenti a supporto della CI/CDE/CD Pipeline

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.

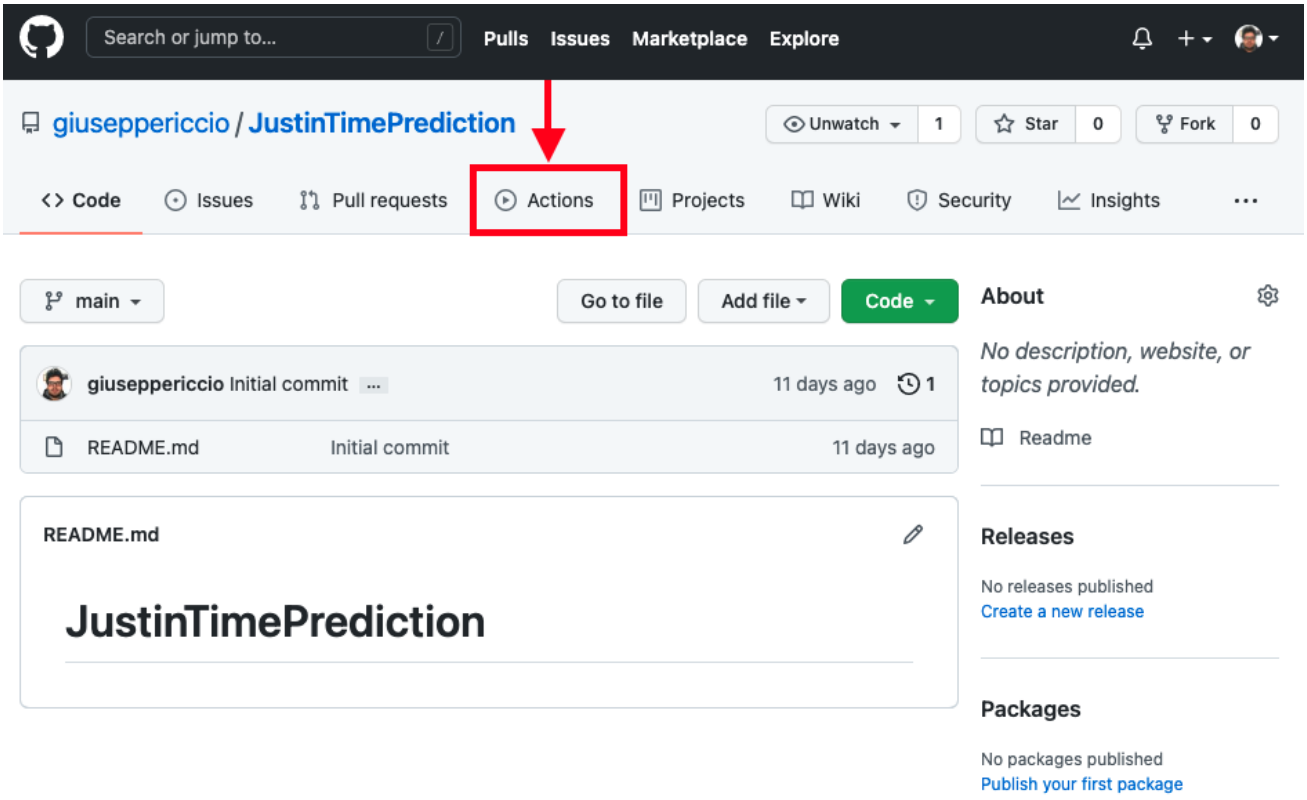


Figura 4: GitHub Actions

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.

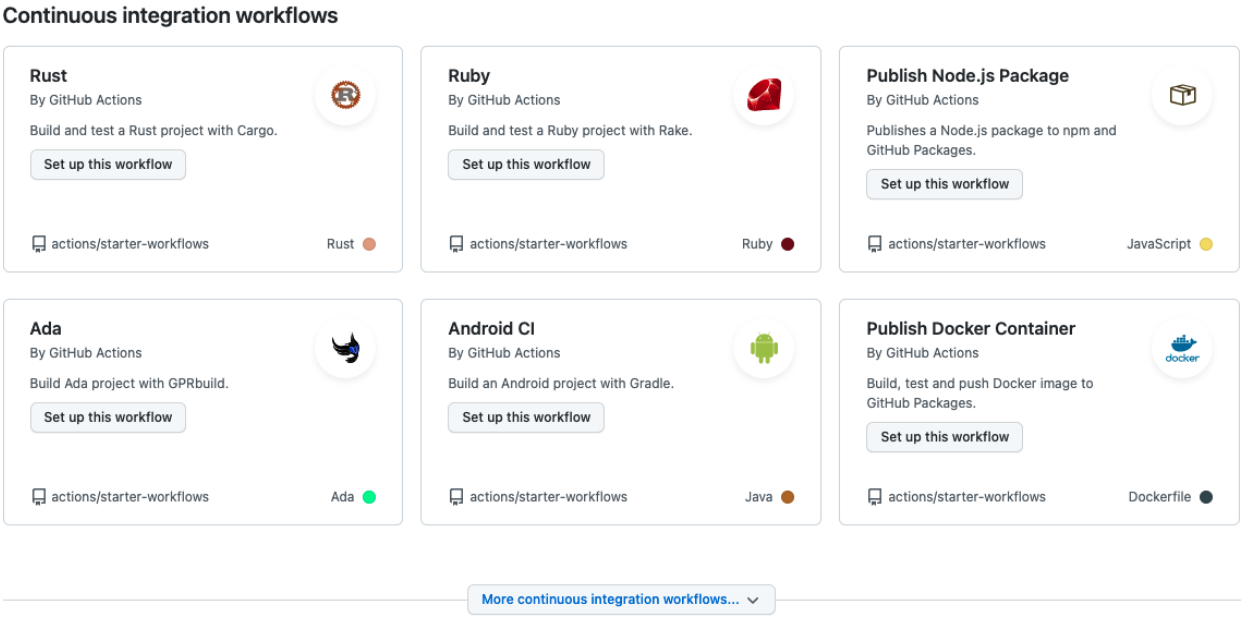


Figura 5: Continuous Integration workflows su GitHub

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.

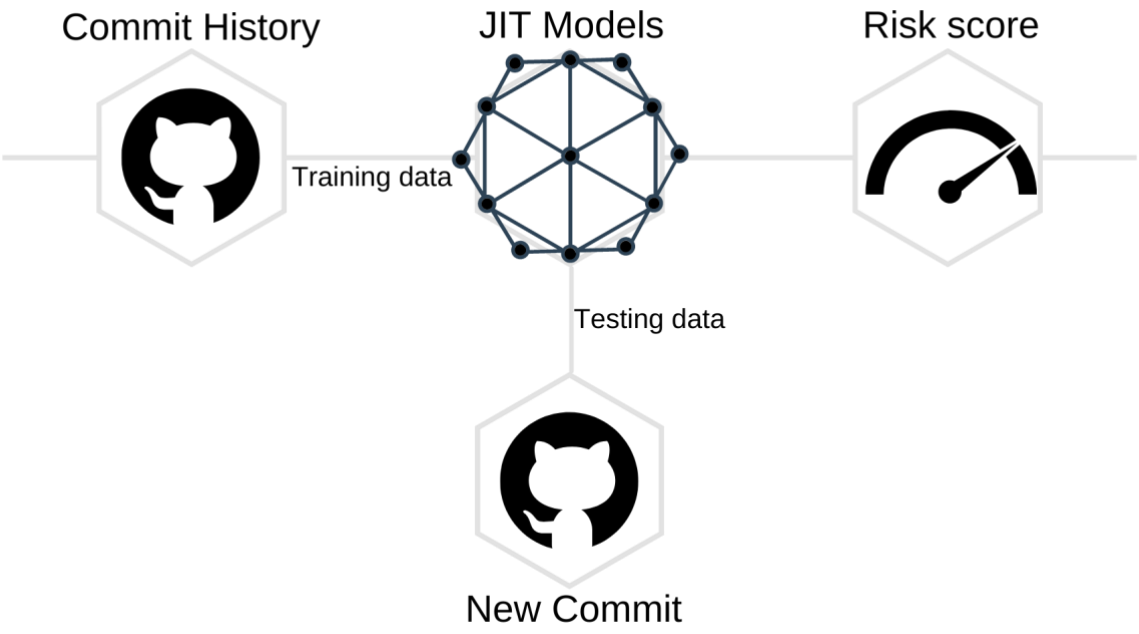


Figura 6: Processo del Just in Time Prediction

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.

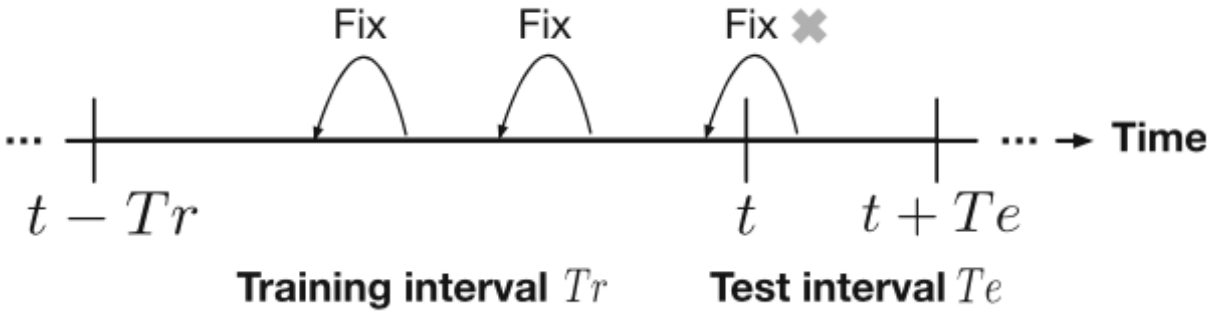


Figura 7: Un esempio di Time Sensitive Change Classification

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.

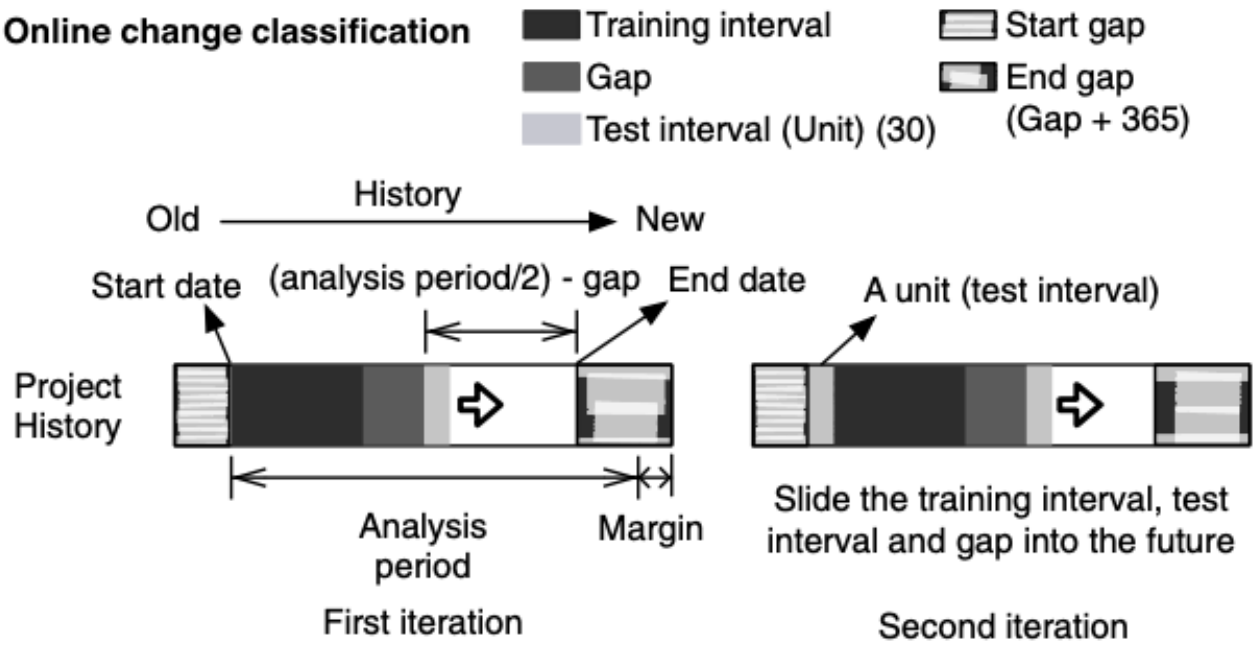


Figura 8: Panoramica dell'Online Change Classification

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:

$$analysis\ period = (CommDatelatest - start\ date) - margin,$$

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

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

dove:

- *CommDatelatest* 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:

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

$$end\ gap = CommDatelatest - end\ date.$$

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.

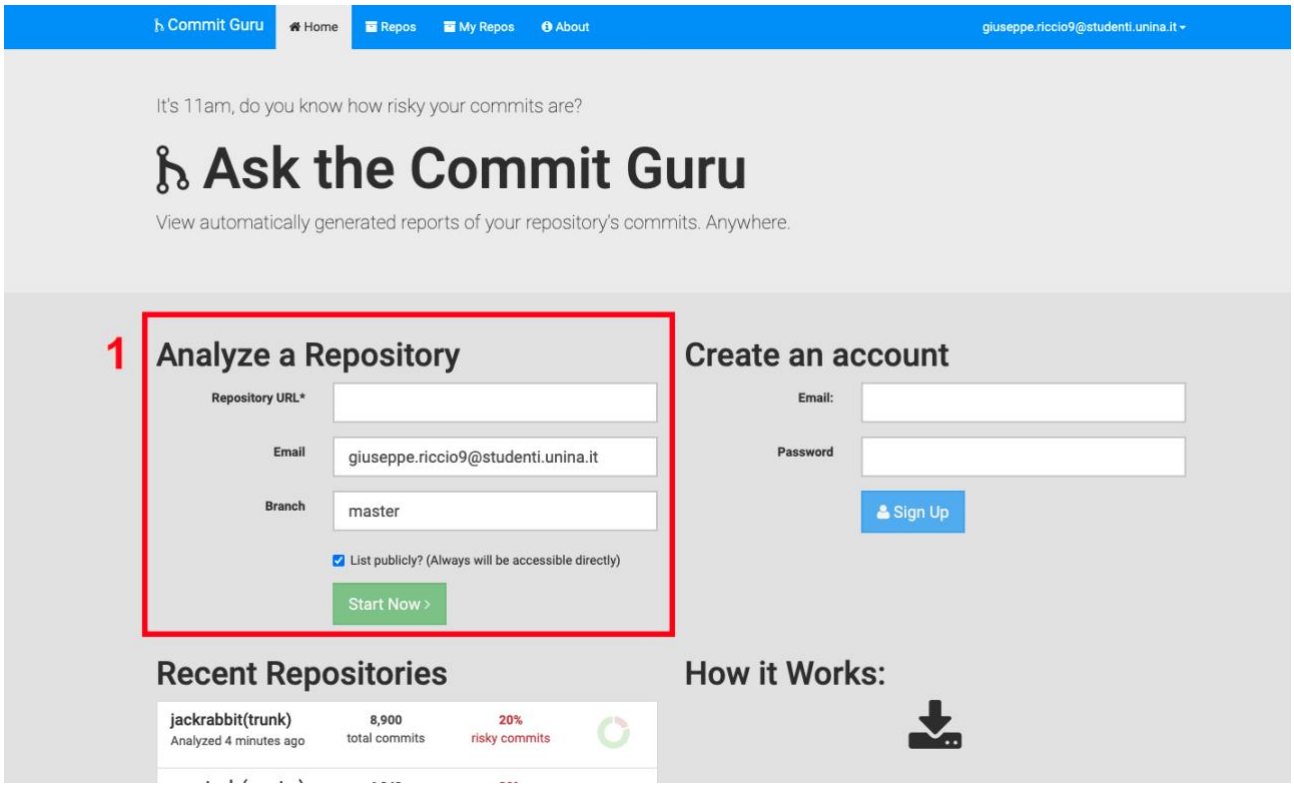


Figura 9: Homepage di Commit Guru

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.

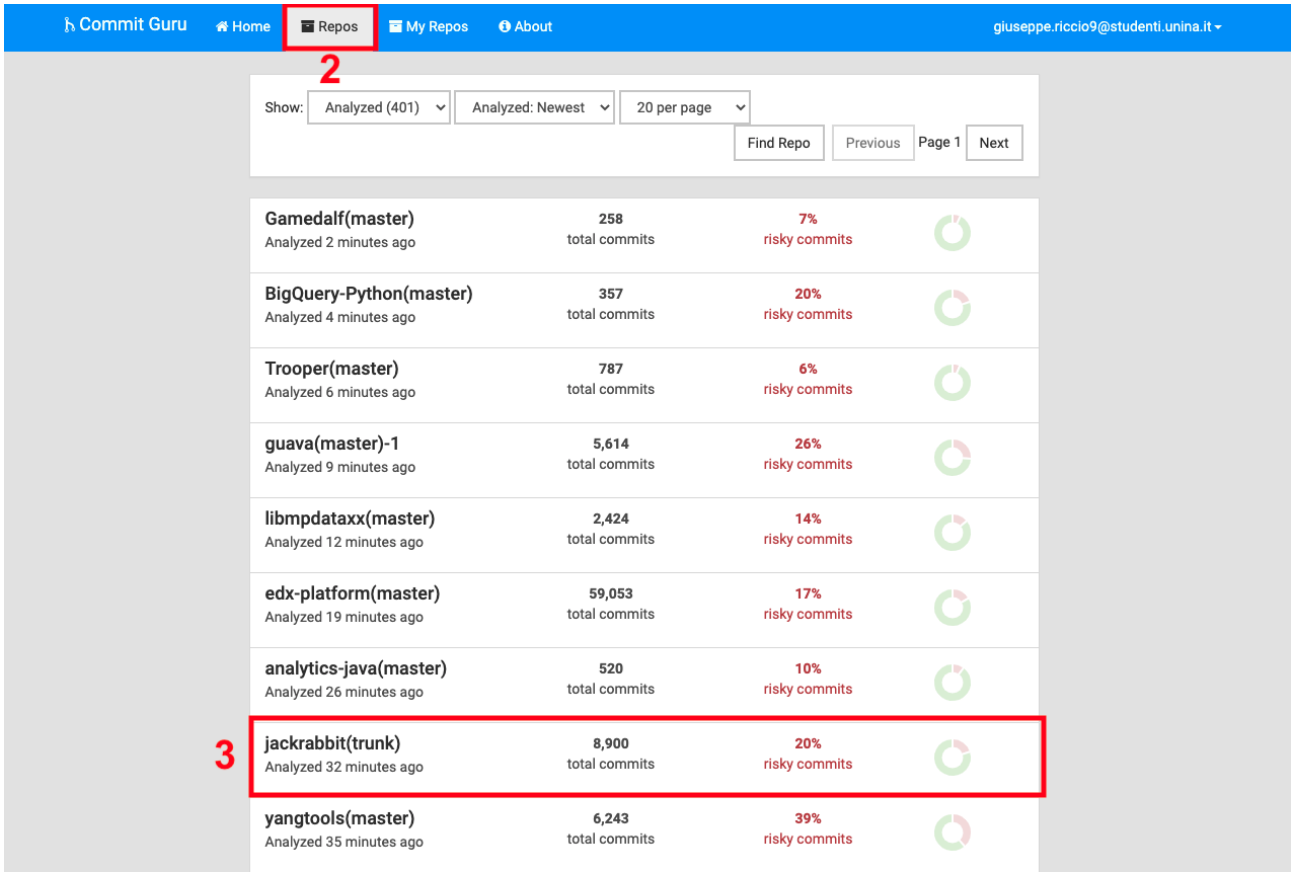


Figura 10: Repositories analizzate da Commit Guru

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.

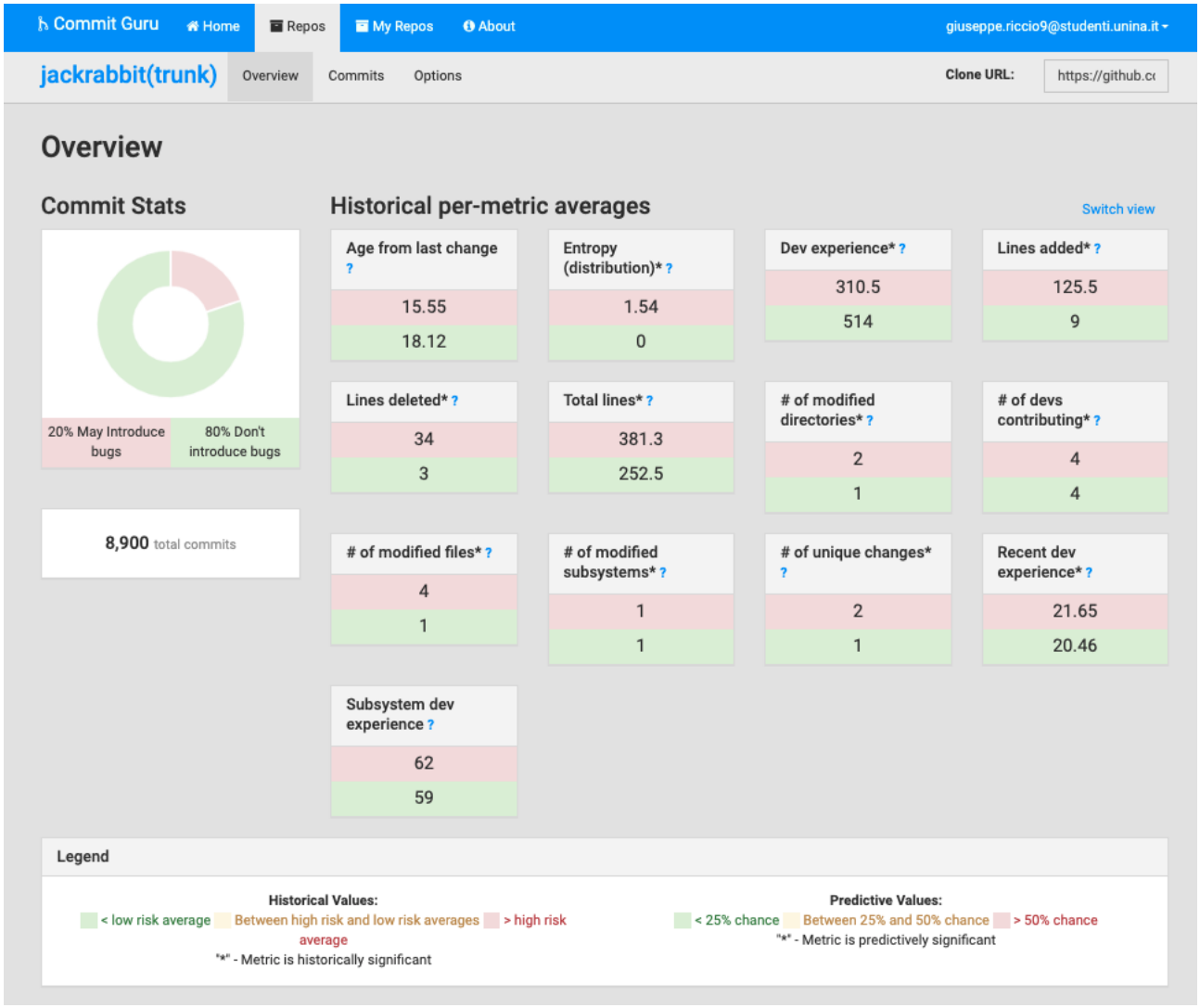


Figura 11: Metriche estratte da Commit Guru

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.

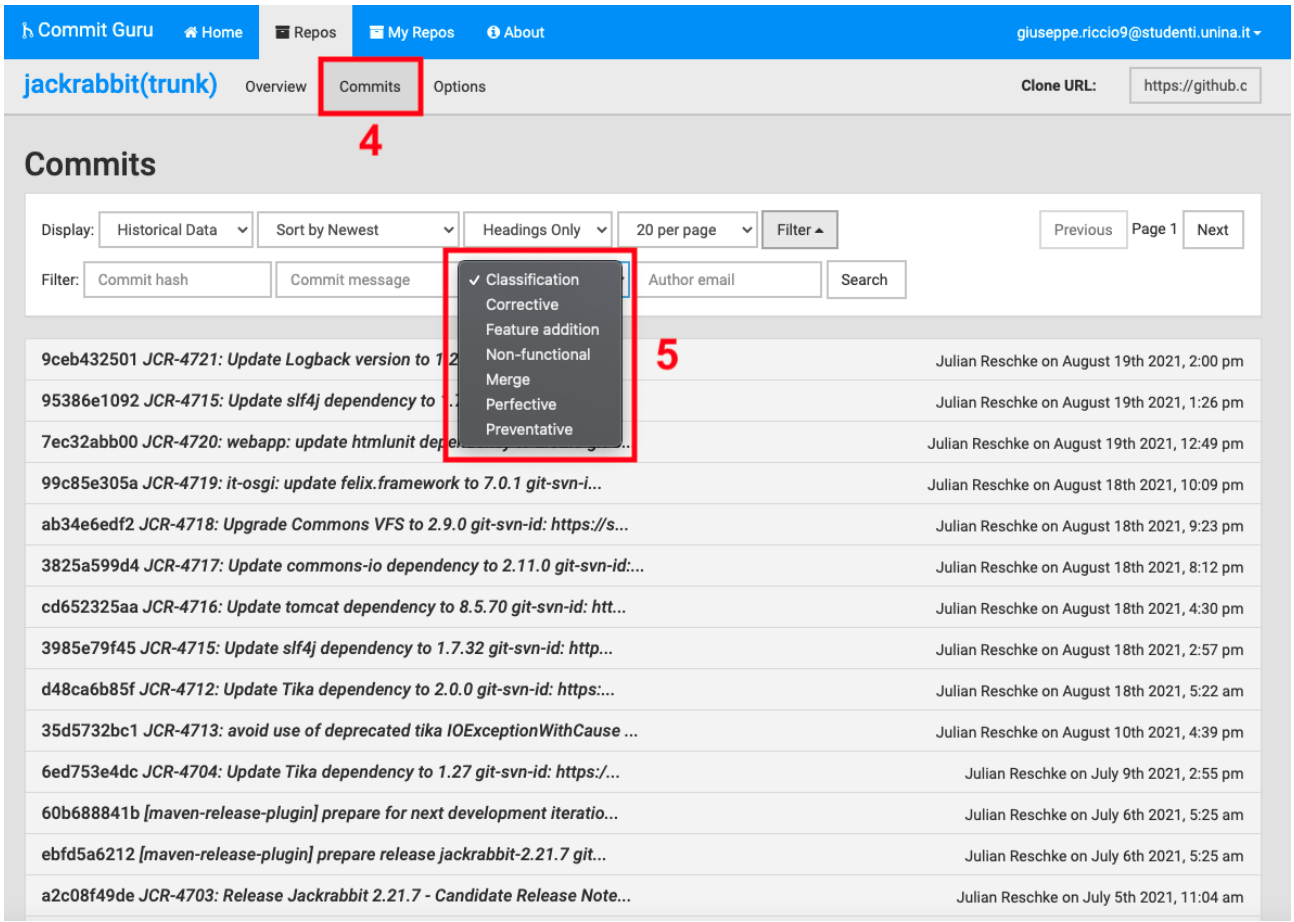


Figura 12: Analisi e classificazione dei singoli commit

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*.

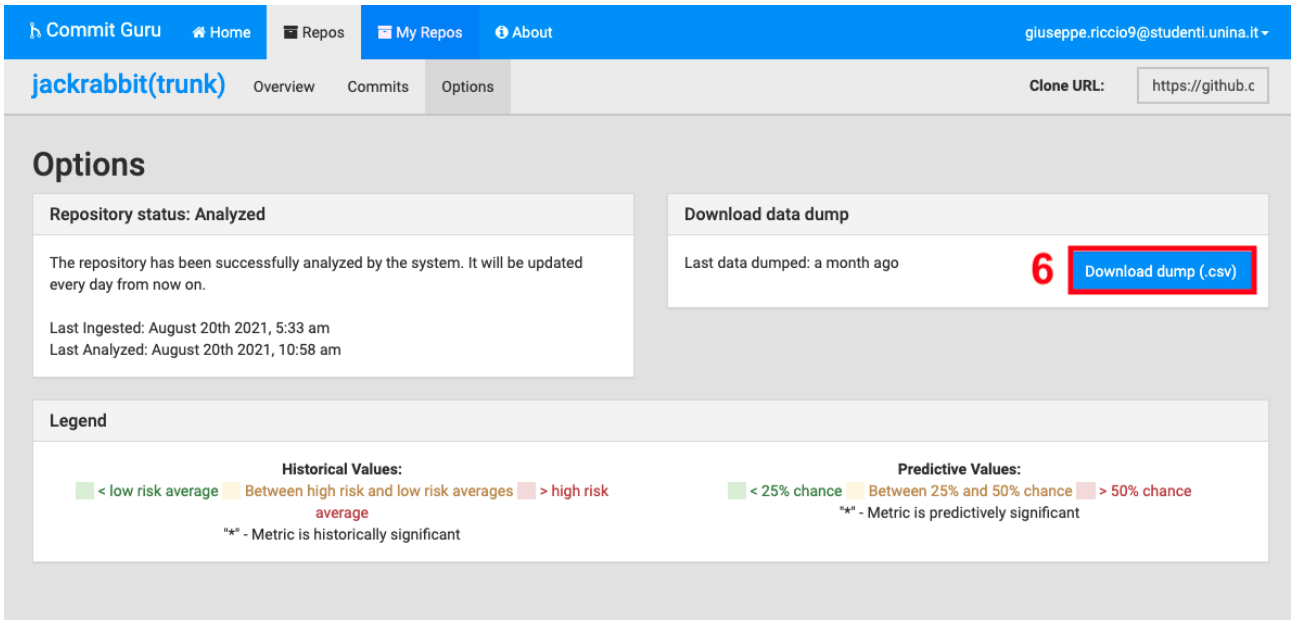


Figura 13: Download del file con le metriche in formato .csv

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.

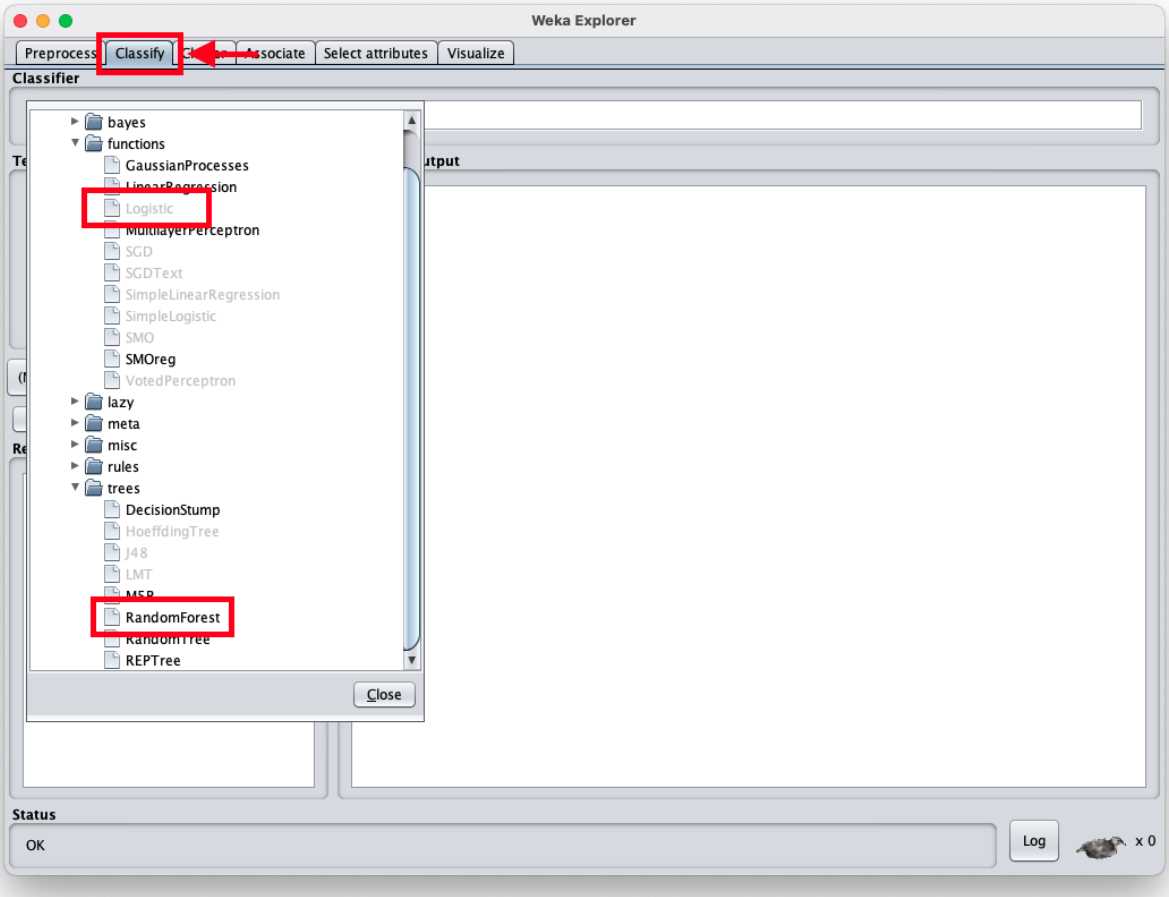


Figura 15: Scelta del modello su Weka

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 [7] 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-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*).

Spesso, i suddetti valori vengono rappresentati in forma matriciale tramite la cosiddetta *"Confusion Matrix"*, che permette di leggere in maniera più rapida ed intuitiva i risultati di una predizione, di seguito viene illustrata una possibile raffigurazione della matrice:

		valore vero		
		p	***n***	**totale**
predizione risultato	***p'***	Vero Positivo (TP)	Falso Positivo (FP)	P'
	n'	Falso Negativo (FN)	Vero Negativo (TN)	N'
	totale	P	N	

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).

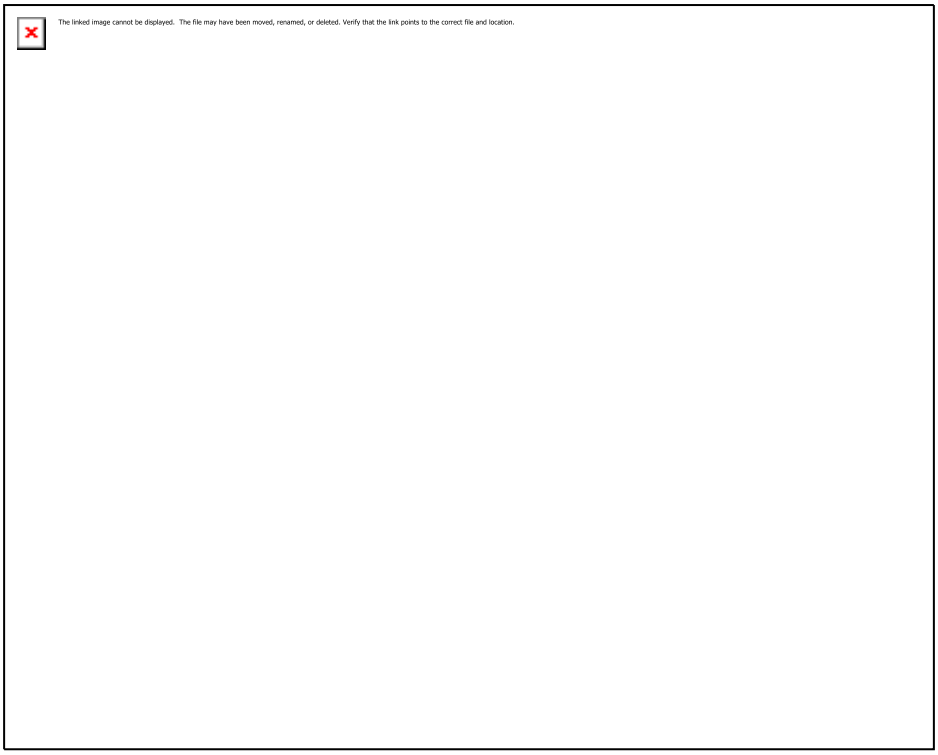


Figura 16: Valori di AUC per particolari curve ROC

Il coefficiente di correlazione di Matthews [8] 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:

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

Equazione 1: Calcolo del MCC

Il punteggio Brier [9] 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).

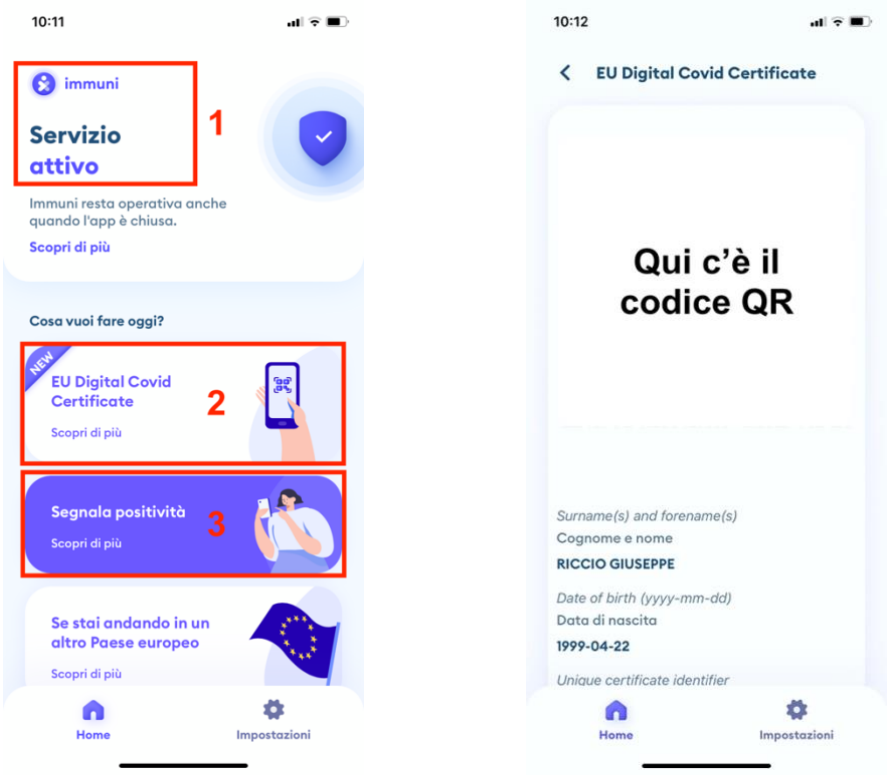


Figura 17: Panoramica app Immuni

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*** [10], 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*** [11], 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).

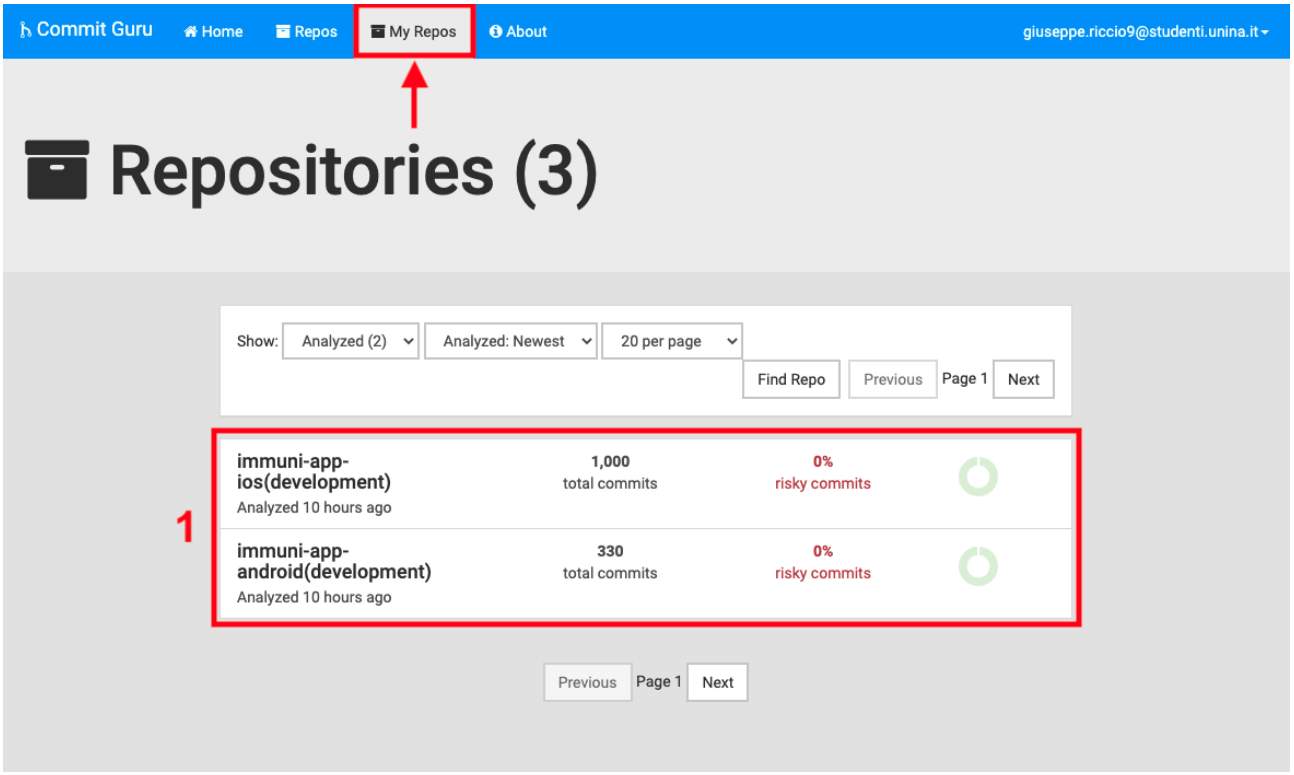


Figura 18: Repositories analizzati da Commit Guru

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.

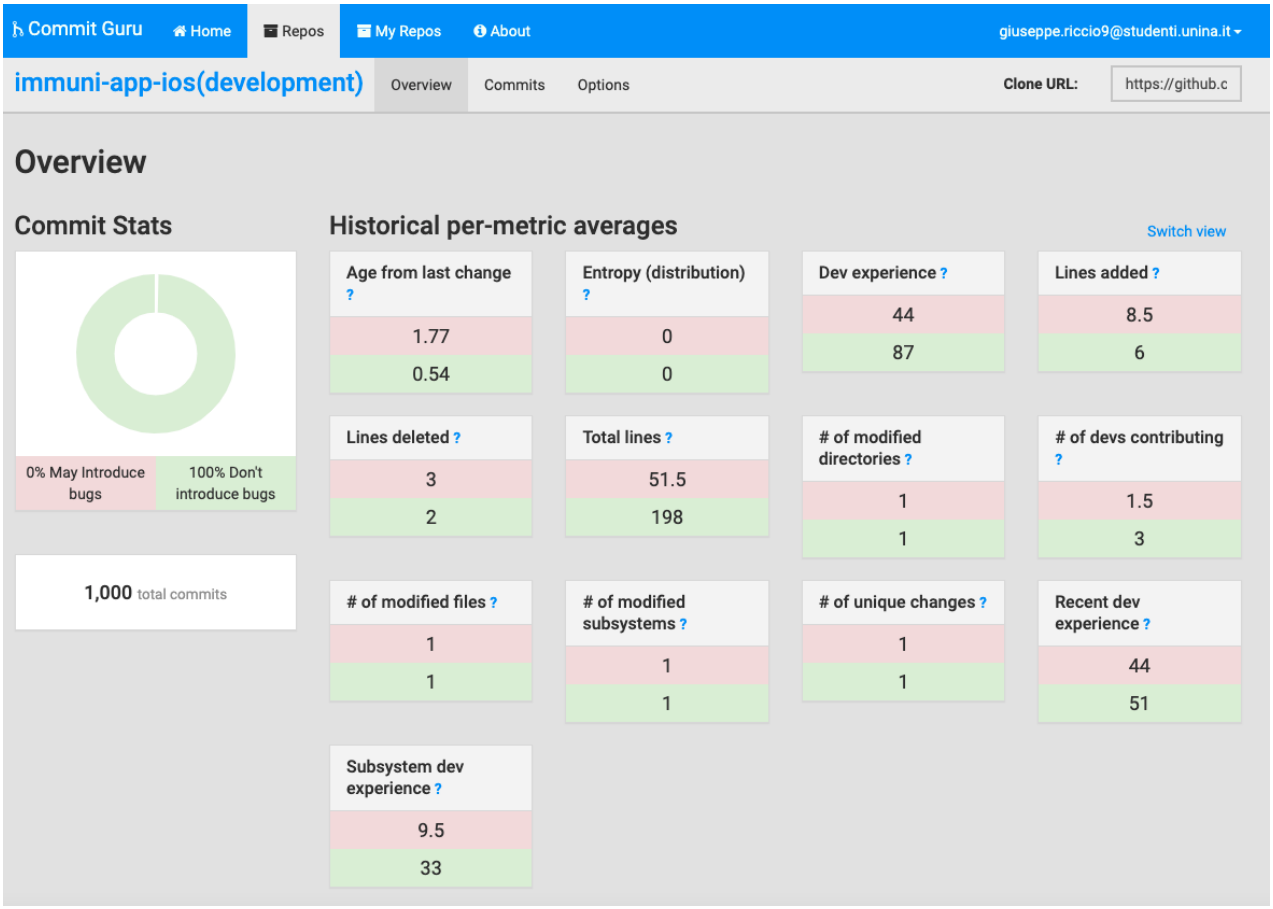


Figura 19: Valori medi delle metriche per Immuni versione iOS

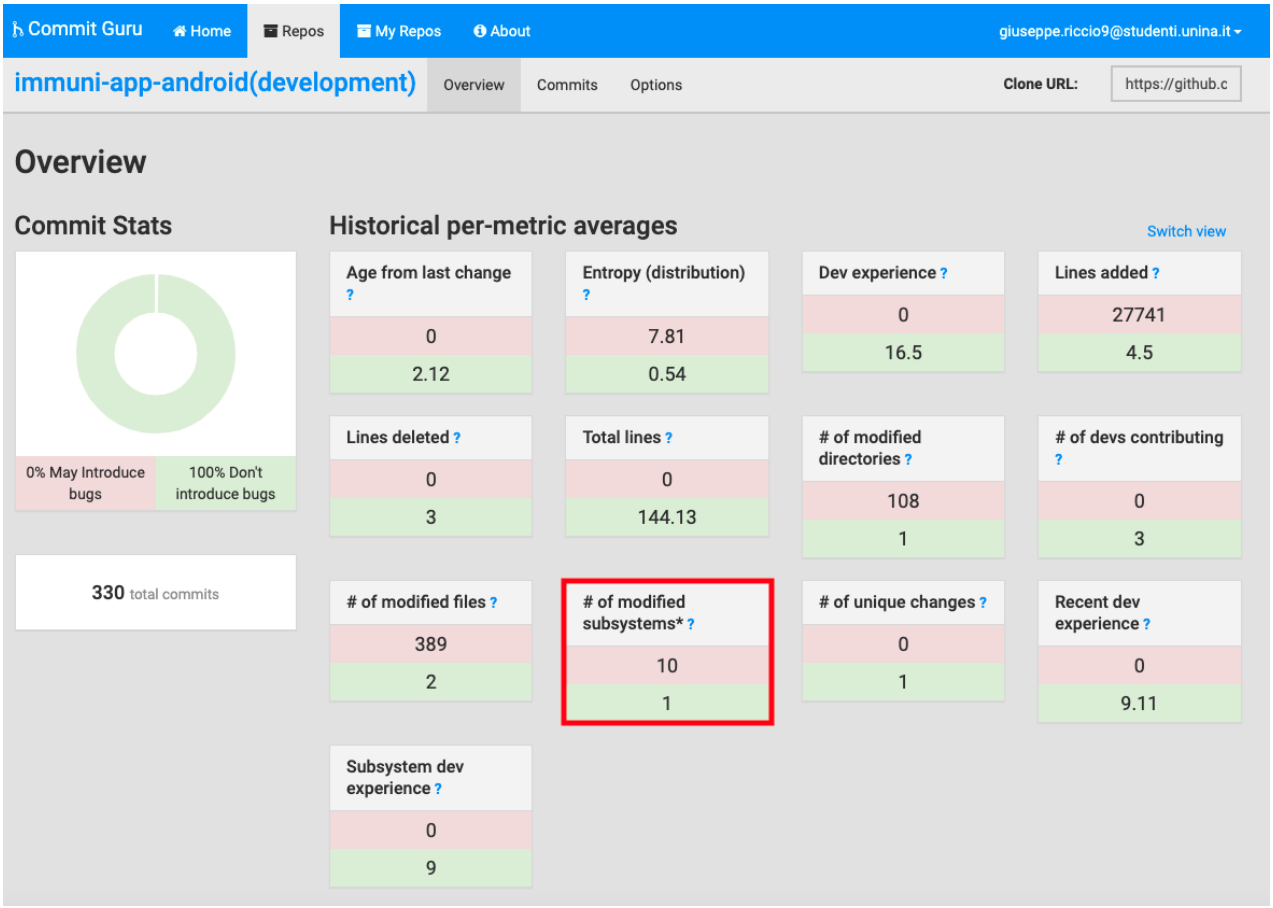


Figura 20: Valori medi delle metriche per Immuni versione Android

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 constanti 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:

$$analysis\ period = (CommDatelatest - start\ date) - margin = 428 - 14 = 414$$

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

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:

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

$$end\ gap = CommDatelatest - end\ date = 1$$

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:

$$analysis\ period = (CommDatelatest - start\ date) - margin = 394 - 14 = 380$$

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

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:

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

$$end\ gap = CommDatelatest - end\ date = 24$$

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

Con i parametri calcolati nel paragrafo precedente possiamo dividere il dataset scaricato da Commit Guru, in formato .csv, in due sottoinsiemi uno per il *training* ed un altro per il *testing*. Il primo dataset ci servirà subito per costruire il modello su Weka, sia nel caso di modello di tipo Random Forest sia nel modello di tipo Logistic Regression.

Di seguito vedremo la costruzione dei modelli, sia per la versione Android che per quella iOS, e le informazioni ottenute da questo passaggio iniziale:

Immuni App Android

1. Random Forest

Il primo modello costruito per la versione Android dell'app Immuni è quello di Random Forest, come è possibile notare dalla Figura 22, questo modello è molto preciso in quanto classifica correttamente tutte le istanze del dataset (1). Inoltre, la costruzione del modello ci fornisce anche altri indicatori, (2) e (3), che permettono di misurare la performance delle metriche usate con riferimento al modello scelto, come ad esempio i valori di TP e FP. Questi valori come abbiamo visto nel paragrafo 3.2.3 ci permettono di costruire i grafici di ROC e MCC.

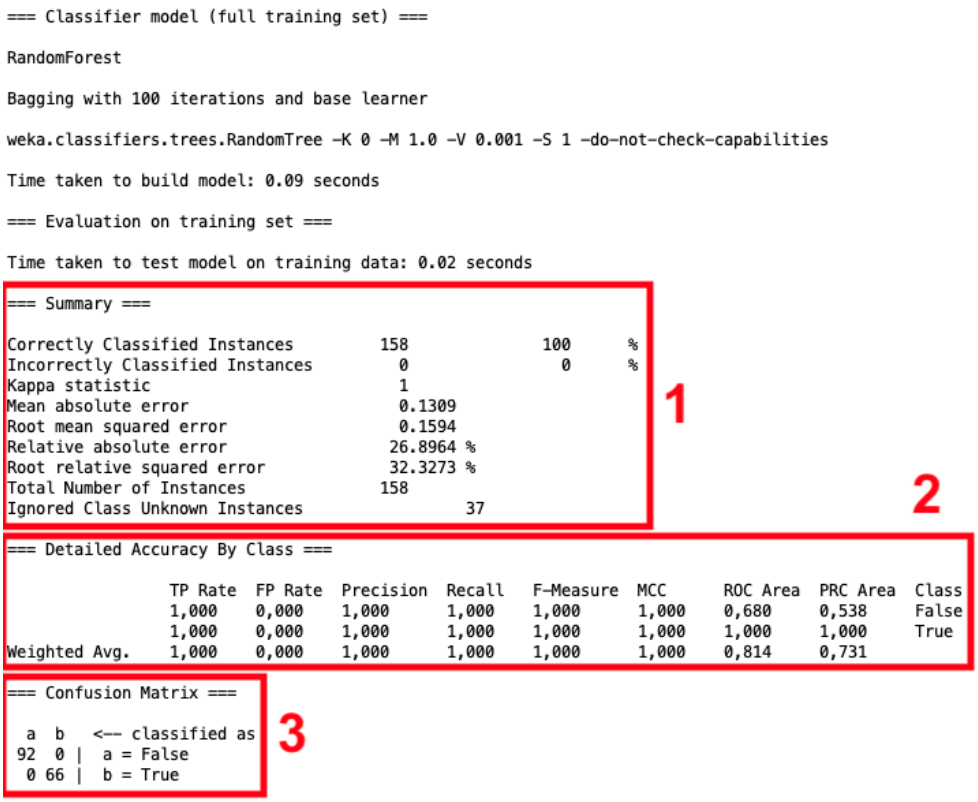


Figura 22: Modello predittivo Random Forest di Immuni versione Android

2. Logistic Regression

Il secondo modello costruito con il training dataset di Immuni versione Android è quello di Logistic Regression, questo modello nel nostro caso è meno preciso del Random Forest, infatti, abbiamo che su 158 istanze quelle classificate correttamente sono 125 e quindi, le altre 33 sono classificate in maniera errata. Questo significa che, come si evince dalla Figura 23 (3) e (4), abbiamo una maggior probabilità di ottenere dei falsi positivi e/o dei falsi negativi il che influisce sulla bontà delle predizioni effettuate un modello di questo tipo, come vedremo nel paragrafo successivo.

Nel modello Logistic Regression oltre alle informazioni riguardo alla classificazione delle istanze, abbiamo anche delle informazioni (1) e (2) riguardo al peso delle metriche sui risultati della classificazione. In particolare, abbiamo che ad esempio, la metrica NS ha un coefficiente di -1.8249, analizzando nel dettaglio il significato di questo valore, il segno negativo sta ad indicare che quella metrica impatta in maniera minore sul risultato, mentre, il valore di 1.8249 indica che quella metrica ha un'importanza maggiore rispetto ad una metrica con valore minore, quindi, NS ha maggiore importanza rispetto a tutte le altre metriche. Per quanto riguarda, l'*"Odds Ratios"*, questo valore è ottenuto a partire dai coefficienti utilizzandoli come argomento della funzione esponenziale *e(x)*.

```
=== Classifier model (full training set) ===

Logistic Regression with ridge parameter of 1.0E-8
Coefficients...
                Class
Variable        False
====================
ns            -1.8249
nd             1.0859
nf            -0.8956
entropy        1.1571
la             0.0016
ld            -0.0024
lt            -0.0056
ndev           0.0633
age           -0.0015
nuc            -0.207
exp            0.0278
rexp           0.0175
sexp          -0.0441
Intercept      2.3962

Odds Ratios...
                Class
Variable        False
====================
ns             0.1612
nd             2.9621
nf             0.4084
entropy        3.1808
la             1.0016
ld             0.9976
lt             0.9944
ndev           1.0654
age            0.9985
nuc             0.813
exp            1.0282
rexp           1.0176
sexp           0.9569
```

1

2

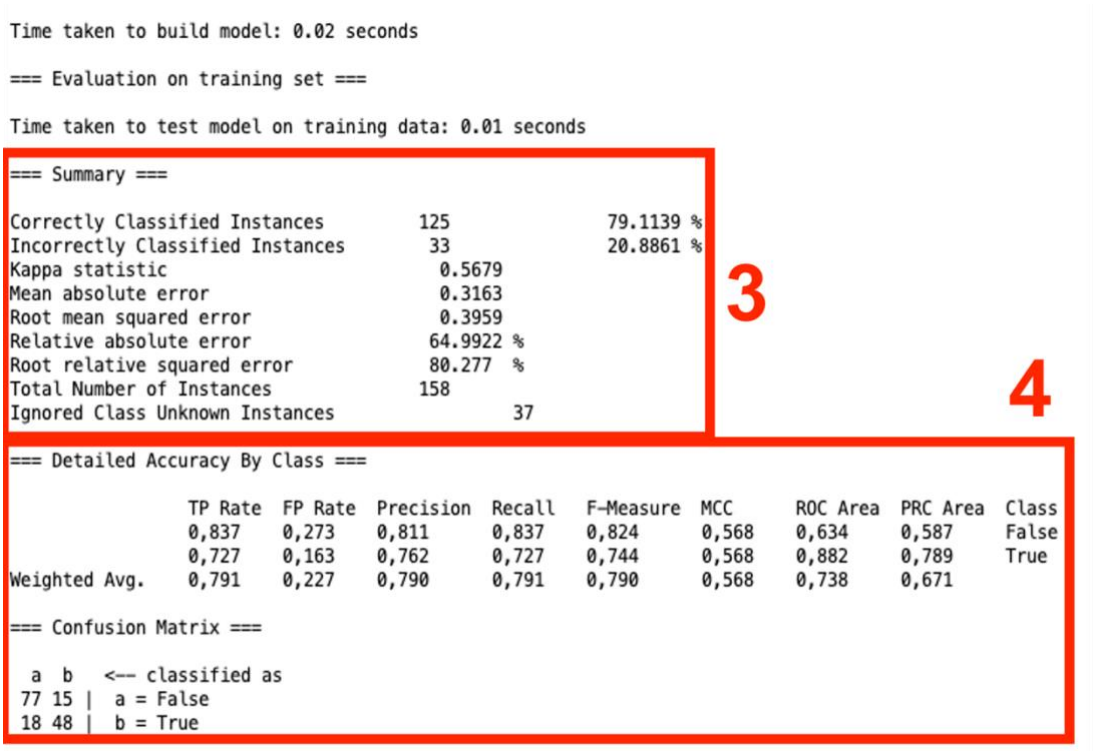


Figura 23: Modello predittivo Logistic Regression di Immuni versione Android

Immuni App iOS

1. Random Forest

Come nel caso precedente, costruiamo il modello Random Forest della versione iOS dell'app Immuni ed ancora una volta, come è possibile notare dalla Figura 24, questo modello è molto preciso in quanto classifica correttamente 395 istanze del dataset su 396 totali (1). Inoltre, i valori di FP e TP (2) e (3), che permettono di misurare la performance delle metriche usate con riferimento al modello scelto hanno dei valori molto elevati, ciò dimostra come questo modello funzioni molto bene per la classificazione binaria (*vera o falsa*) come nel nostro caso di studio.

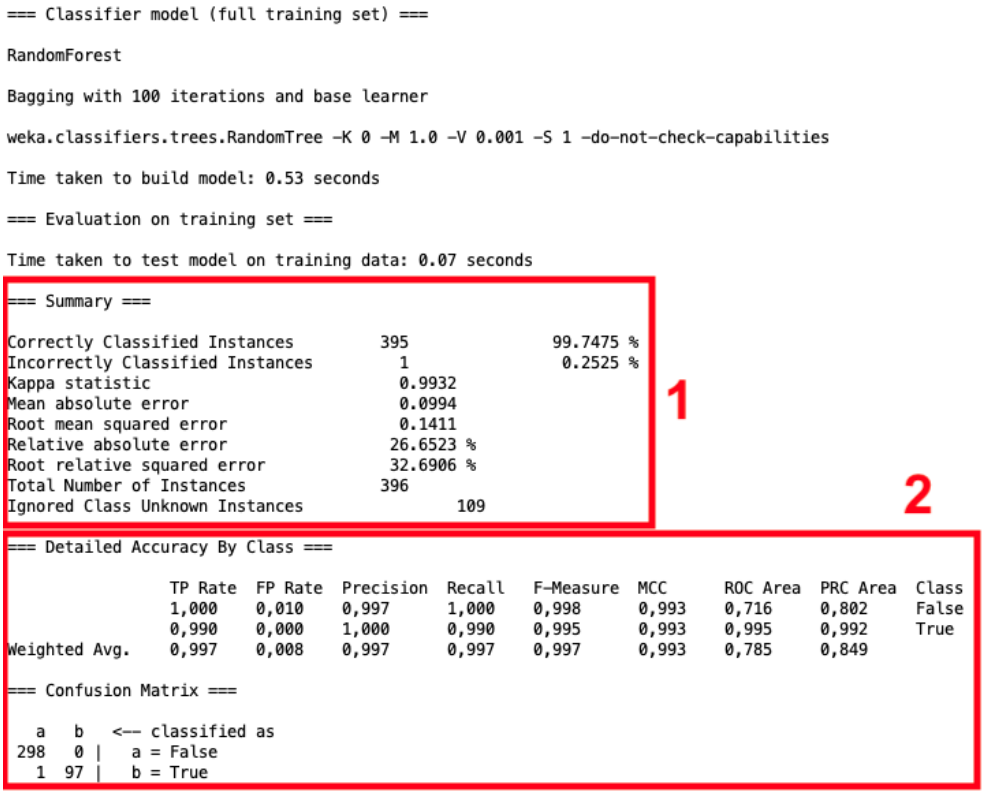


Figura 24: Modello predittivo Random Forest di Immuni versione iOS

2. Logistic Regression

Il secondo modello costruito con il training dataset di Immuni versione iOS è quello di Logistic Regression, questo modello risulta meno preciso del Random Forest, infatti, abbiamo che su 396 istanze quelle classificate correttamente sono 301 e quindi, le altre 95 sono classificate in maniera errata. Questo significa che, come si evince dalla Figura 25 (3) e (4), abbiamo dei valori di falsi positivi e/o di falsi negativi più bassi del modello precedente, che porta ad ottenere delle predizioni sbagliate.

Dopo aver analizzato le informazioni che permettono di valutare e classificare le istanze del dataset, possiamo osservare le informazioni (1) e (2) che riguardano il peso delle metriche sui risultati della classificazione. Ad esempio, la metrica ENTROPY ha un coefficiente di 1.4721, analizzando nel dettaglio il significato di questo valore, abbiamo che il segno positivo sta ad indicare che quella metrica impatta in maniera maggiore sul risultato, mentre, l'ampiezza di questa metrica indica che essa ha un'importanza maggiore rispetto a tutte le altre metriche, ma non prevale in maniera netta su quest'ultime a differenza della versione Android.

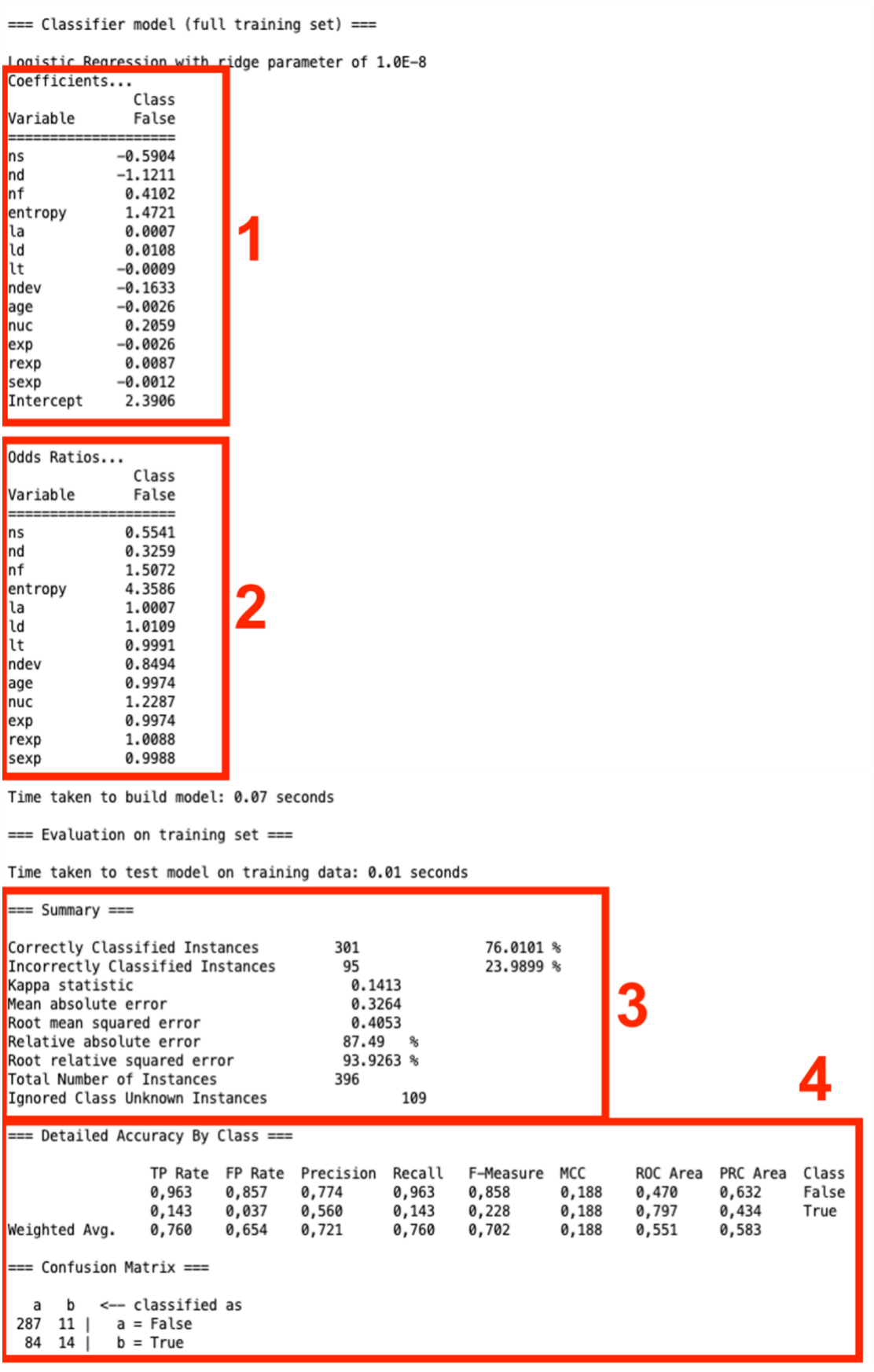


Figura 25: Modello predittivo Logistic Regression di Immuni versione iOS

4.1.3 Validazione del modello

Per la validazione dei modelli costruiti nel paragrafo precedente, utilizziamo come anticipato, un sottoinsieme dei dataset che chiamiamo di *training*.

Questi dati sono presi nell'arco temporale di 30 giorni come da calcoli effettuati nel paragrafo 4.1.1, e ci permettono di verificare se il modello costruito effettua delle predizioni più o meno corrette e quali sono le sue performance in termini di precisione.

Di seguito si propongono i risultati della validazione, per le due versioni dell'app Immuni:

Immuni App Android

1. Random Forest

Come è possibile notare dalla Figura 26 (1), il modello predittivo Random Forest riesce a predire in maniera corretta circa il 78% delle istanze all'interno del *training dataset*, il che ci fornisce delle buone performance in termini di *errore assoluto medio* che è pari a 0.3667, ricordando che tale valore è tanto migliore quanto più si avvicina allo zero.

```
=== Classifier model (full training set) ===

RandomForest

Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.06 seconds

=== Evaluation on test set ===

Time taken to test model on supplied test set: 0 seconds

=== Summary ===

Correctly Classified Instances           7               77.7778 %
Incorrectly Classified Instances         2               22.2222 %
Kappa statistic                          0.3571
Mean absolute error                      0.3667
Root mean squared error                  0.4027
Relative absolute error                 80.6107 %
Root relative squared error             87.5747 %
Total Number of Instances                9

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,857    0,500    0,857      0,857    0,857      0,357    0,571     0,826     False
               0,500    0,143    0,500      0,500    0,500      0,357    0,571     0,625     True
Weighted Avg.  0,778    0,421    0,778      0,778    0,778      0,357    0,571     0,781

=== Confusion Matrix ===

 a b   <-- classified as
 6 1 | a = False
 1 1 | b = True
```

1

2

Figura 26: Testing del modello predittivo Random Forest di Immuni versione Android

2. Logistic Regression

Per quanto riguarda il modello predittivo Logistic Regression, a differenza di quanto notato durante la sua costruzione, esso è leggermente migliore come correttezza nella predizione, infatti, come si evince dalla Figura 27 (1) abbiamo una percentuale di istanze correttamente predette pari circa all'89%.

Conseguentemente anche il valore dell'*errore assoluto medio* è più basso del caso precedente, ed in particolare, pari a 0.2221 che rappresenta un valore molto buono.

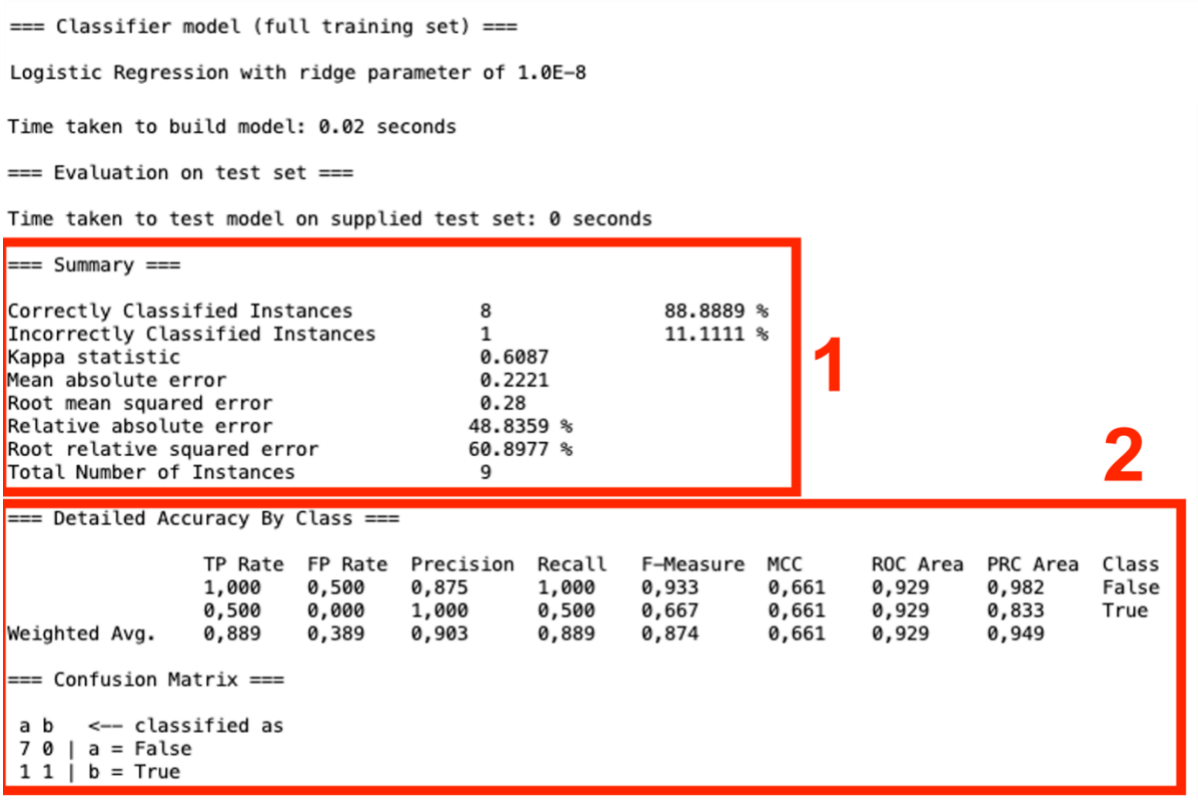


Figura 27: Testing del modello predittivo Logistic Regression di Immuni versione Android

Immuni App iOS

1. Random Forest

Il modello predittivo Random Forest per la versione iOS dell'app Immuni, come si nota nella Figura 28, ha delle prestazioni molto basse ed una percentuale di istanze correttamente predette solo del 33% (a fronte della controparte Android che aveva una percentuale del 78%). Ciò impatta anche sul valore dell'*errore assoluto medio* che è pari a 0.4926, quindi è quasi equamente probabile che il valore predetto sia errato o meno.

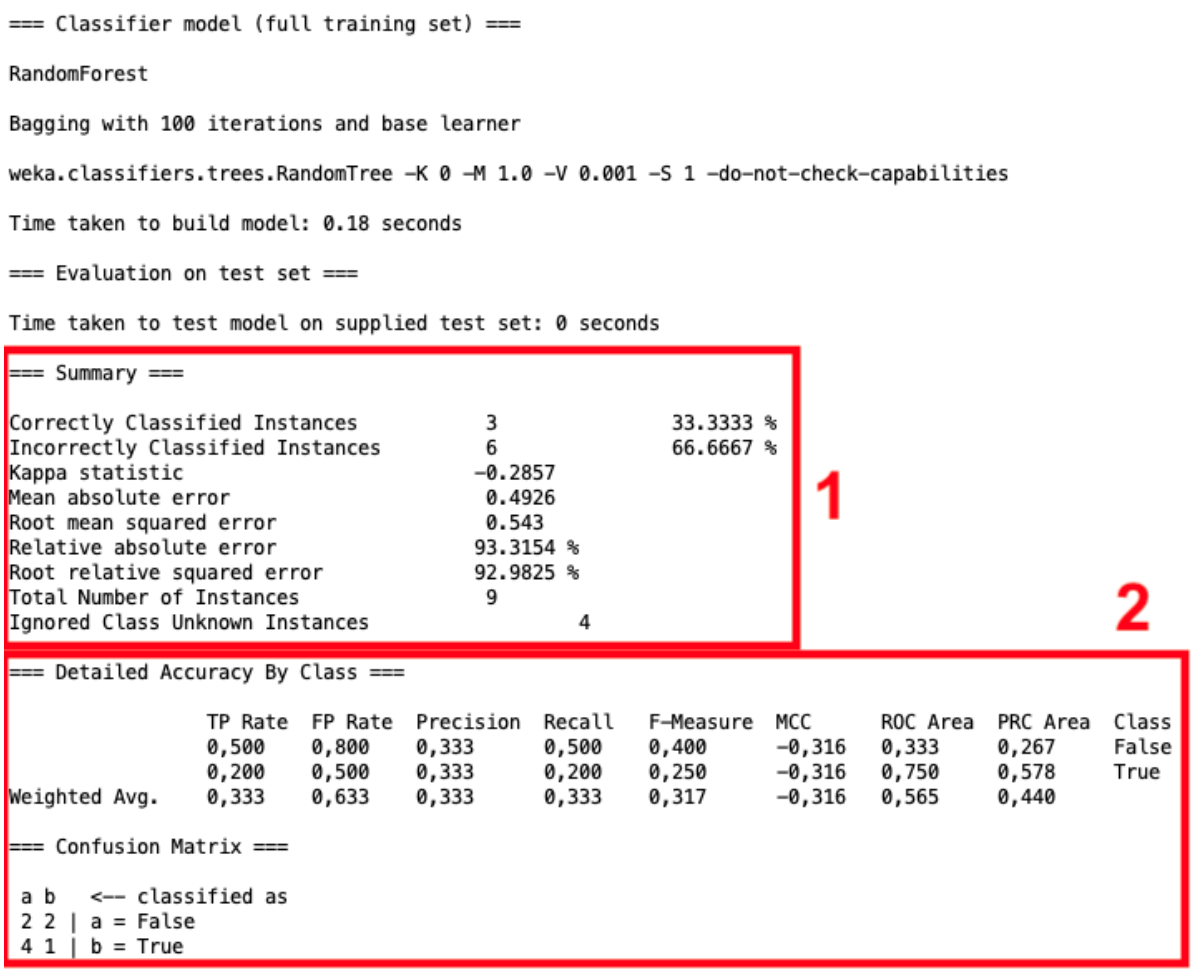
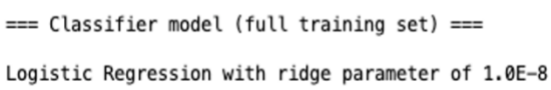


Figura 28: Testing del modello predittivo Random Forest di Immuni versione iOS

2. Logistic Regression

Come nel modello precedente, anche nel Logistic Regression per la versione iOS dell'app Immuni otteniamo dei risultati poco soddisfacenti in termini di predizione, ottenendo nuovamente una percentuale del solo 33% di istanze predette in maniera corretta (a differenza dell'89% ottenuto nella versione Android). Il valore di *errore assoluto medio* è di 0.499, quindi, anche in questo caso regna la totale incertezza sulle predizioni effettuate.



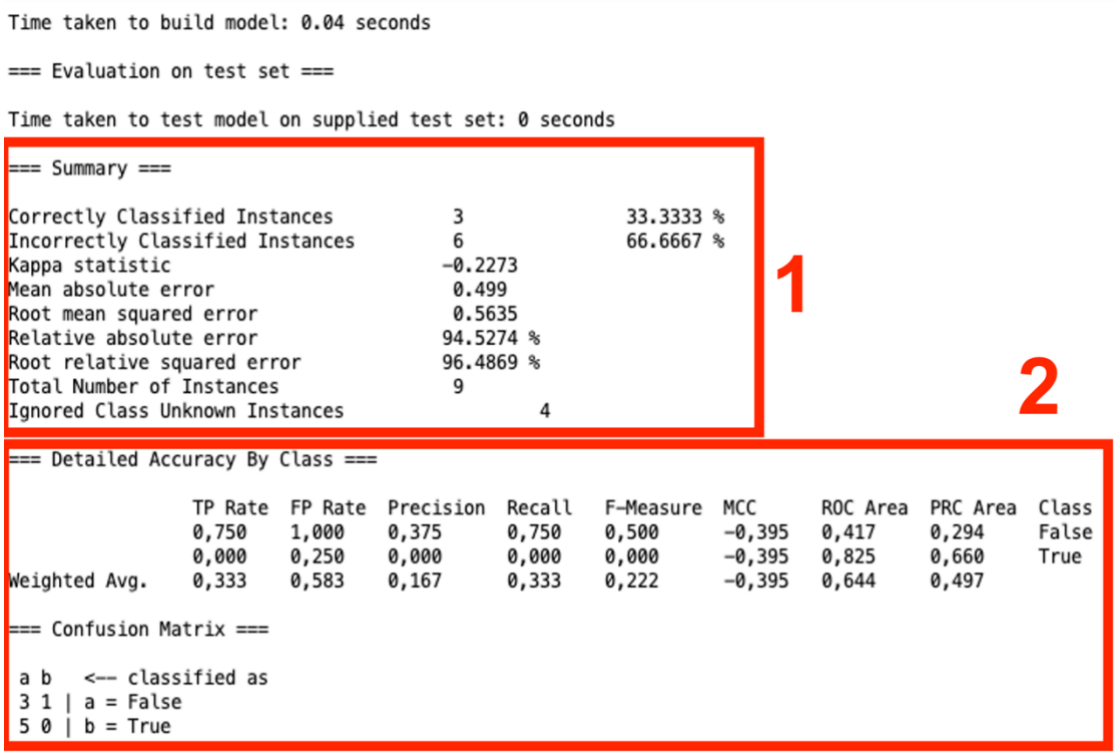


Figura 29: Testing del modello predittivo Logistic Regression di Immuni versione iOS

4.1.4 Risultati ed osservazioni

Misurazione delle prestazioni nella costruzione del modello

1. Curva ROC (Receiver Operating Characteristic) e AUC (Area Under the Curve)

Utilizzando i dati forniti da Weka nella costruzione dei due modelli per l'app Immuni versione Android (Figura 22 e 23), possiamo subito notare come il Random Forest sia un modello "ideale" per questo tipo di classificazione, infatti abbiamo che il TP Rate, ovvero il tasso di veri positivi, raggiunge subito il massimo che corrisponde alla situazione in cui la classificazione è stata corretta per tutte le istanze contenute nel dataset di *training data* forniti per la costruzione del modello. Nella situazione mostrata nel Grafico 1, quindi, abbiamo che l'area AUC sottesa dalla curva del modello RF è pari ad 1, altresì il modello Logistic Regression presenta una curva caratteristica ROC che non raggiunge il massimo, ma si assesta intorno al valore di 0.84 con un'area sottesa dalla curva di 0.738.

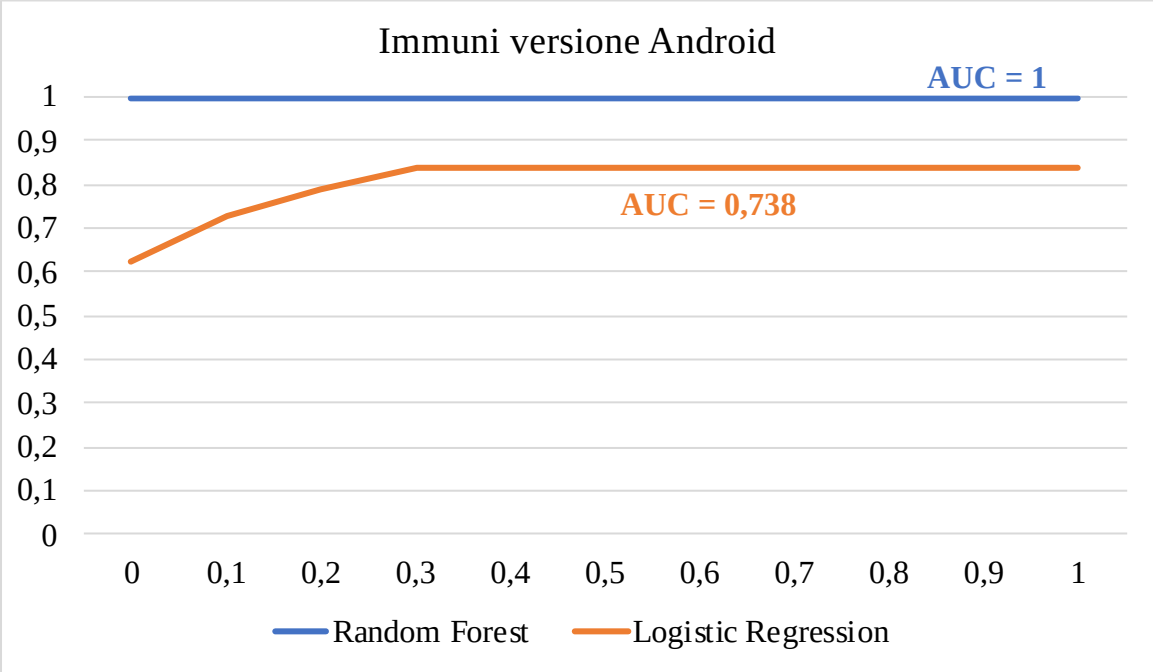


Grafico 1: Curva ROC e AUC per la costruzione del modello (versione Android)

Nel caso della versione iOS (Figura 24 e 25), si ottengono dei risultati simili a quelli del caso precedente ma con valori leggermente diversi e con un divario tra i due modelli meno netto di prima, come si evince dal Grafico 2. Infatti, abbiamo che il modello RF in questo caso tende al valore massimo più dolcemente, e lo raggiunge in corrispondenza del valore di FP Rate (tasso di falsi positivi) di 0.1, ottenendo un'area AUC di 0.995, mentre, per il modello LR abbiamo un valore a regime di circa 0.97 ed un'area di 0.797.

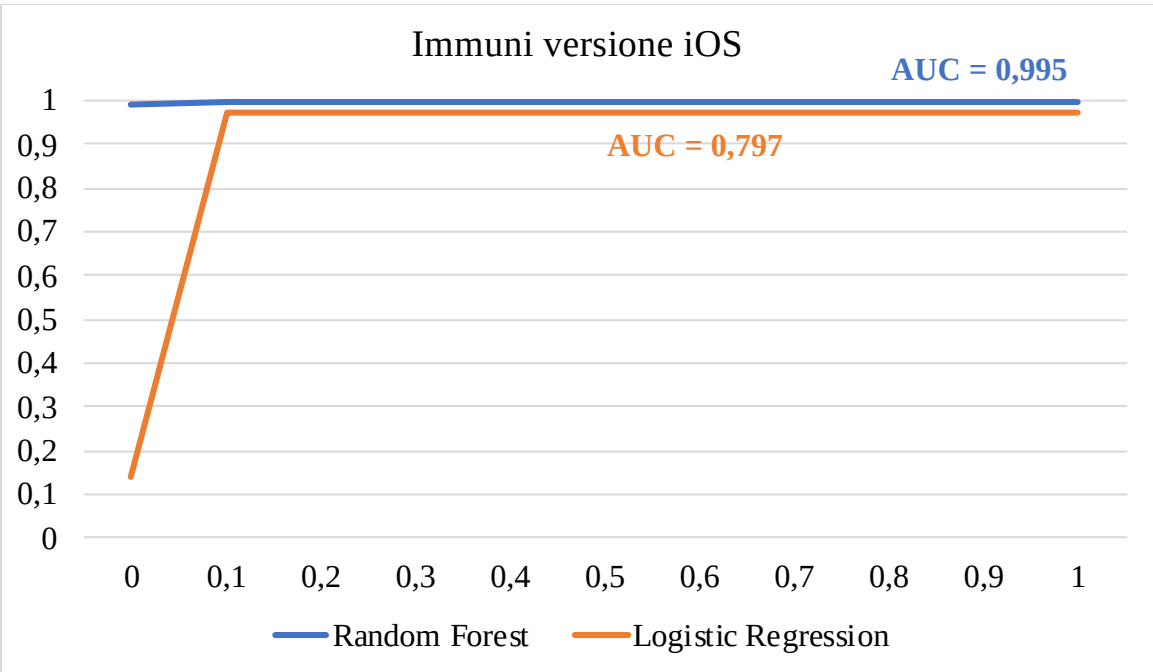


Grafico 2: Curva ROC e AUC per la costruzione del modello (versione iOS)

2. Matthews Correlation Coefficient

Il valore di MCC per entrambe le versioni (iOS e Android) di Immuni è calcolato direttamente da Weka, infatti, abbiamo che:

- **versione Android:** per il modello RF abbiamo un valore di MCC pari ad 1, che è un valore ottimale in quanto rappresenta la condizione di perfetta classificazione delle istanze del dataset. Nel caso, invece, del modello LR il valore di MCC è di 0.568 il che indica una maggiore incertezza sulla precisione dei risultati ottenuti dalla classificazione fornita da tale modello.
- **versione iOS:** per il modello RF abbiamo un valore di MCC pari a 0.993, che è un buon risultato e permette di ottenere dei risultati con un buon grado di precisione nella classificazione del dataset. Altresì, il modello LR ha un valore di MCC pari solo a 0.188, in questo caso, abbiamo un alto grado di incertezza che non ci permette di essere precisi nei risultati.

3. Brier score

Il Brier score rappresenta l'errore quadratico medio e ci permette di capire quanto è grande la differenza tra il valore predetto ed il valore vero, ovviamente minore è questo numero migliore sono i risultati che otteniamo. Nel nostro caso avremo che:

- **versione Android:** per il modello RF abbiamo un Brier score di 0.1594 e, quindi, in questo caso avremo una classificazione molto precisa. Mentre, nel modello LR il Brier score è di 0.3959, un valore più alto del precedente e questo comporta una minore precisione, anche se, tale valore è ancora accettabile.
- **versione iOS:** per il modello RF abbiamo un Brier score di 0.1411 il che ci permette di affermare che i valori predetti sono molto vicini ed in alcuni casi uguali ai valori veri. Al contrario, nel modello LR il Brier score è di 0.4053, questo valore introduce una maggiore incertezza nei valori predetti e non ci permette, quindi, di effettuare delle affermazioni preventive sui risultati.

Misurazione delle prestazioni nella validazione del modello

1. Curva ROC (Receiver Operating Characteristic) e AUC (Area Under the Curve)

Utilizzando i dati forniti da Weka nella validazione dei due modelli per l'app Immuni versione iOS (Figura 26 e 27), possiamo subito notare come il Logistic Regression, a differenza di quanto visto per la sua costruzione, rappresenti il modello migliore in termini di precisione, infatti abbiamo che il TP Rate, ovvero il tasso di veri positivi, raggiunge il massimo in corrispondenza di un FP Rate di 0.5 e, quindi, da questo punto in poi permette di classificare correttamente tutte le istanze contenute nel *testing dataset*. Questa condizione è mostrata nel Grafico 3, che ci permette anche di determinare l'area AUC sottesa dalla curva del modello LR che è pari a 0.929, invece, il modello Random Forest presenta una curva caratteristica ROC che non raggiunge il massimo, ma raggiunge un valore max di circa 0.89 con un'area sottesa dalla curva di soli 0.571.

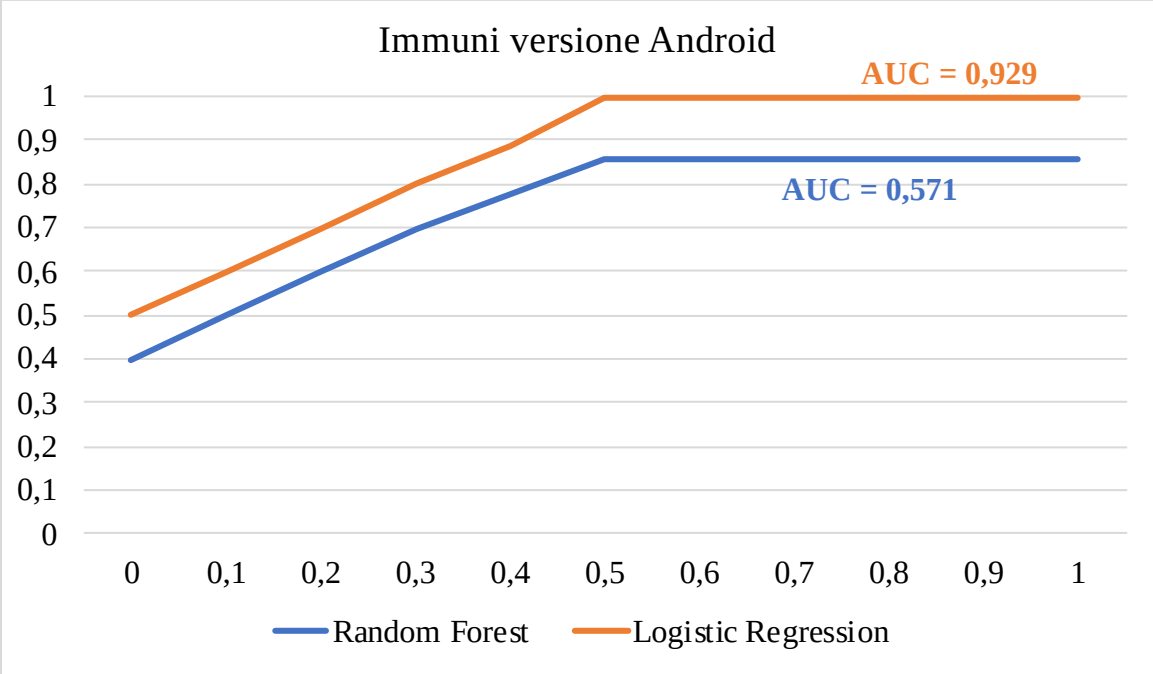


Grafico 3: Curva ROC e AUC per la validazione del modello (versione Android)

Nel caso della versione iOS (Figura 28 e 29), per entrambi i modelli si ottengono dei risultati molto bassi, il che è sintomo o di un'errata scelta dei modelli predittivi oppure di un problema di classificazione troppo complesso, tutto ciò influisce sicuramente sui risultati delle predizioni che risultano poco precisi. Infatti, abbiamo che il modello RF in questo caso non supera il valore di 0.5, ottenendo un'area AUC di 0.333, mentre, per il

modello LR abbiamo un valore in corrispondenza del FP Rate pari ad 1 di 0.75 con un'area AUC pari a 0.417. Questi valori di AUC sono addirittura inferiori alla *chance diagonal* che individua un'area AUC di 0.5 e rappresenta la condizione necessaria e sufficiente affinché le predizioni effettuate siano accettabili.

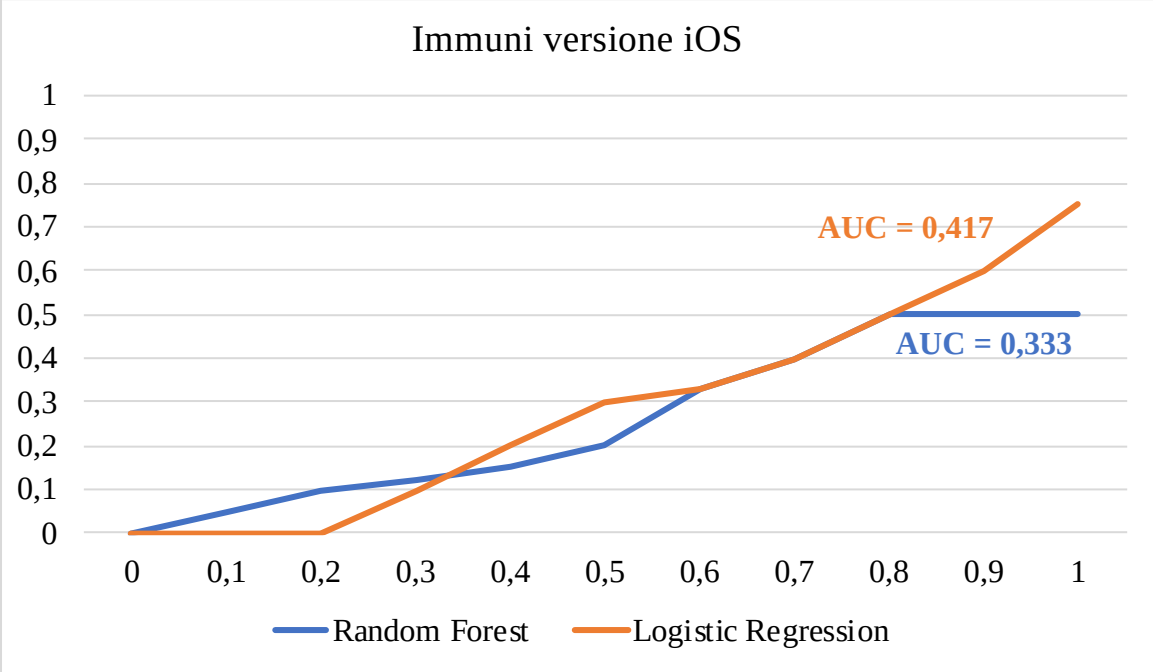


Grafico 4: Curva ROC e AUC per la validazione del modello (versione iOS)

2. Matthews Correlation Coefficient

Il valore di MCC, riguardo alla validazione dei modelli, per entrambe le versioni (iOS e Android) di Immuni calcolati da Weka sono riportati di seguito:

- **versione Android:** per il modello RF abbiamo un valore di MCC pari a 0.357, che è un valore piuttosto basso e rappresenta una condizione in cui i valori predetti sono affetti da un alto grado di incertezza. Nel caso, invece, del modello LR il valore di MCC è di 0.661 il che indica una maggiore precisione nei risultati ottenuti dalla predizione effettuata con tale modello.
- **versione iOS:** per il modello RF abbiamo un valore di MCC pari a -0.316, ricordando che il valore di MCC può variare tra -1 ed 1, questo valore è diretta conseguenza delle cattive prestazioni che questo modello ottiene nella predizione sul nostro *testing dataset*. Allo stesso modo, il modello LR ha un valore di MCC pari solo a -0.395, ed anche in questo caso, abbiamo un alto grado di incertezza sui risultati.

3. Brier score

Il Brier score che, come detto in precedenza, rappresenta l'errore quadratico medio nella validazione dei modelli predittivi creati con Weka, assume i seguenti valori:

- **versione Android:** per il modello RF abbiamo un Brier score di 0.4027 e, quindi, come abbiamo avuto già modo di dire con tale modello otteniamo dei risultati meno precisi. Mentre, nel modello LR il Brier score è di 0.28, un valore più basso del precedente, il che ci permette di ricavare delle predizioni più corrette e con minore incertezza.
- **versione iOS:** per il modello RF abbiamo un Brier score di 0.543, tale valore rappresenta una condizione di totale indeterminatezza sulla correttezza delle predizioni. In modo del tutto speculare, anche il modello LR presenta un Brier score molto basso e pari a solo 0.5635, questo valore non permette di ottenere dei risultati accettabili, in quanto essi assumono dei valori completamente aleatori.

Conclusioni

Come abbiamo avuto modo di capire durante la trattazione di questo elaborato, lo sviluppo software è diventata un'attività cruciale per molte aziende e per questo occorre garantire durante tale processo un certo grado di qualità e sicurezza.

Inoltre, con l'evoluzione dei cicli di vita del software con particolare interesse verso le cosiddette *metodologie agili* che permettono di focalizzare lo sviluppo sulla parte implementativa piuttosto che quella progettuale, che viene svolta in maniera ristretta sulla particolare funzionalità oggetto di rilascio, quest'ultima casistica rientra in pieno in un contesto di sviluppo *Continuous Integration*.

Abbiamo visto che, proprio con l'uso delle *Continuous Practices,* nasce l'esigenza di avere delle tecniche per prevedere quando un particolare rilascio (*commit*) è difettoso o meno, perché essendo questi rilasci sempre più frequenti è più facile introdurre dei bug all'interno del codice, ma correggendoli in maniera preventiva rispetto al rilascio permette di contenere i danni economici sul sistema software stesso.

In questo contesto, abbiamo studiato le tecniche di predizione 'Just-in-Time' che attraverso l'estrazione di particolari metriche da un log di commit, presi in particolari intervalli temporali da questo deriva il termine "time" nella tecnica studiata, permette di costruire dei modelli predittivi che permettono di classificare i commit come difettosi o meno.

In particolare, abbiamo preso in considerazione due modelli predittivi, il primo è il

Random Forest che attraverso il Machine Learning permette di costruire degli alberi di apprendimento, che vengono poi usati per effettuare la predizione dei commit. Il secondo modello è quello del Logistic Regression, anch'esso sfrutta tecniche dell'Intelligenza Artificiale, in cui le probabilità che un determinato evento accada vengono usate come input di attivazione della funzione logistica (*sigmoide*).

Applicando questi due modelli al nostro caso di studio, come si è evinto dai risultati mostrati nel paragrafo 4.1.4, il modello di Random Forest è risultato molto preciso nella sua costruzione con valori di AUC, MCC e Brier score molto buoni, il che faceva presumere che tale modello fosse quello migliore per effettuare le predizioni dei commit difettosi.

Tuttavia, nella versione Android dell'app Immuni si è avuto modo di capire che il modello migliore per effettuare la predizione è quello del Logistic Regression, mentre, nella versione iOS addirittura entrambi i modelli sono risultati insufficienti ad ottenere dei risultati accettabili nella classificazione dei commit.

Quindi, in definitiva, la tecnica di predizione del 'Just-in-Time' si è mostrata come uno strumento molto potente in grado di prevenire situazioni di malfunzionamento del software, ma che allo stesso tempo richiede particolare attenzione nella scelta del modello predittivo usato. Infatti, abbiamo potuto vedere come nel nostro caso lo stesso modello, Logistic Regression, abbia prestazioni diverse sulla medesima app scritta per due sistemi operativi differenti.

TUTTO IL MATERIALE USATO IN QUESTO ELABORATO LO TROVATE SUL MIO GITHUB

SE IL QR CODE NON FUNZIONA, IL LINK AL REPOSITORY È IL SEGUENTE: https://github.com/giuseppericcio/JustinTimePrediction

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: 18 agosto 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: 19 agosto 2021

[6] Waikato University, *"Weka 3: Machine Learning Software in Java"*, https://www.cs.waikato.ac.nz/ml/weka/, Consultato: 19 agosto 2021

[7] Wikipedia, *"Receiver operating characteristic"*, https://it.wikipedia.org/wiki/Receiver_operating_characteristic, Consultato: 20 agosto 2021

[8] Wikipedia, *"Matthews correlation coefficient"*, https://en.wikipedia.org/wiki/Matthews_correlation_coefficient, Consultato: 20 agosto 2021

[9] Wikipedia, *"Brier score"*, https://en.wikipedia.org/wiki/Brier_score, Consultato: 20 agosto 2021

[10] Wikipedia, *"Kotlin (linguaggio di programmazione)"*, https://it.wikipedia.org/wiki/Kotlin_(linguaggio_di_programmazione), Consultato: 23 agosto 2021

[11] Apple, *"Swift. Un linguaggio potente e aperto a tutti per creare fantastiche app."*, https://www.apple.com/it/swift/, Consultato: 23 agosto 2021