UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

TESI DI LAUREA

Flakiness Detection: An Extensive Analysis

RELATORE

Prof. Fabio Palomba

Dott. Valeria Pontillo

Università degli Studi di Salerno

CANDIDATO

Angelo Afeltra

Matricola: 0522501354

Anno Accademico 2022-2023

Questa tesi è stata realizzata nel

Dedica o citazione

Abstract

In un sistema software i casi di test assicurano che le modifiche effettuate al codice non vadano ad influire negativamente sulle funzionalità già esistenti, tant'è vero che quando gli sviluppatori apportano modifiche al loro codice, eseguono il cosiddetto test di regressione per rilevare se le modifiche effettuate introducono eventuali bug all'interno del sistema. Tuttavia, alcuni test potrebbero essere flaky, ovvero non deterministici è assumere sia un comportamento di pass che di failure quando vengono eseguiti sullo stesso codice. Inizialmente la flakiness è stata riconosciuta andando a eseguire più volte il caso di test e osservando un cambiamento nel comportamento del caso di test. Tale approccio è dispendioso in termini di tempo e costoso dal punto di vista computazionale, pertanto oggi si studiano tecniche per prevedere la flakiness, e.g., machine learning..
Numerosi studi hanno esaminato vari dataset e seguito metodologie specifiche per esplorare l'efficacia dell'uso del machine learning nell'affrontare la flakiness dei test. Tuttavia, molti di questi si sono concentrati maggiormente sulla fattibilità dell'approccio piuttosto che sulla sua utilità pratica, utilizzando principalmente approcci in-vitro.

Con questo lavoro di tesi si vuole invertire la rotta, analizzando come lavoro un approccio in-vivo di predizione della flakiness basato sul machine learning. L'obiettivo è capire se la predizione alla flakiness rappresenta un'alternativa valida al ReRun della test suite N volte.

Per raggiungere questo scopo, abbiamo generato due dataset, noti come FlakeFlagger e IDoFT, utilizzando le metriche statiche proposte da Pontillo et al. [1] e il primo di questi ci consente di condurre un confronto diretto con lo stato dell'arte. Su entrambi i dataset, abbiamo innanzitutto identificato la pipeline ottimale per prevedere la flakiness. Successivamente, abbiamo valutato l'efficacia di tale pipeline per un utilizzo Within-Project e Cross-Project.

I risultati ottenuti non solo ci hanno permesso di superare lo stato dell'arte nel dataset FlakeFlagger, ottenendo un notevole F1-Score dell'80%, ma abbiamo anche dimostrato l'utilità di questa pipeline nell'ambito di progetti individuali. Tuttavia, tale efficacia non si estende a un utilizzo Cross-Project non supervisionato, in cui la mancanza di informazioni sulla repository target rende difficile affrontare i problemi di adattamento del dominio. Al contrario, mediante un approccio Cross-Project supervisionato, siamo stati in grado di ottenere prestazioni paragonabili a quelle raggiunte in un contesto Within-Project.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

La pratica del software testing ha radici che risalgono ai primi sviluppi del software, che si sono verificati dopo la seconda guerra mondiale. Questo processo riveste un'importanza cruciale, poiché è finalizzato a garantire la qualità e l'affidabilità del software stesso. Inizialmente, il metodo principale per il testing era il debugging, ovvero l'individuazione e la correzione dei bug. Tuttavia, negli anni '80, gli sviluppatori hanno iniziato a considerare il testing in una prospettiva più ampia, comprendendolo come un processo di controllo di qualità integrato nell'intero ciclo di vita dello sviluppo del software.

Secondo Alexander Yaroshko, autore di una pubblicazione su uTest, negli anni '90 si è assistito a una transizione dai semplici test a un processo più completo denominato "controllo di qualità". Questo approccio copre l'intero ciclo di sviluppo del software e coinvolge le fasi di pianificazione, progettazione, creazione ed esecuzione dei casi di test, nonché il supporto per i casi di test esistenti e gli ambienti di test.

Oggi, la pratica del software testing può essere definita come l'esecuzione di una serie di test mirati, al fine di identificare difetti, errori o comportamenti indesiderati nel software. Questi test possono essere eseguiti su diverse componenti del software, tra cui il codice sorgente, l'interfaccia utente e le funzionalità generali.

Va notato che nel corso degli anni non solo la tipologia di testing eseguita è

cambiata, con l'integrazione di nuove modalità per garantire una migliore qualità del software, ma anche il momento in cui viene effettuato il testing nel ciclo di sviluppo del software. In passato, il testing era spesso separato dallo sviluppo e veniva eseguito in una fase avanzata del ciclo di vita del software, spesso poco prima del rilascio. Questo approccio ha spesso causato una corsa contro il tempo per individuare e correggere i difetti prima del lancio.

Tuttavia, l'esecuzione di test all'inizio del ciclo di sviluppo, nota come "continuous testing", consente di mantenere il testing come un'attività prioritaria piuttosto che posticiparla. Inoltre, eseguire test in modo precoce riduce i costi associati alla correzione dei difetti. L'approccio di continuous testing implica l'esecuzione automatica dei test in modo continuo durante tutto il ciclo di sviluppo, garantendo che ogni modifica sia sottoposta a una verifica immediata. Questo approccio aiuta a individuare e risolvere i problemi in tempo reale, evitando accumuli o propagazioni di difetti nelle fasi successive.

I vantaggi del software testing sono molteplici e vanno oltre la semplice individuazione dei difetti. Contribuisce al miglioramento generale della qualità del software, garantendo il corretto funzionamento e la conformità alle specifiche, riducendo al minimo i malfunzionamenti e i comportamenti inattesi. Inoltre, ottimizza l'allocazione delle risorse, individuando tempestivamente i problemi e prevenendo costosi ritardi e correzioni post-lancio.

Un aspetto fondamentale è anche la sicurezza del software. Il testing può individuare vulnerabilità e falle di sicurezza, consentendo agli sviluppatori di adottare misure preventive per proteggere i dati e prevenire minacce informatiche. In un mondo sempre più orientato alla privacy e alla sicurezza, il testing svolge un ruolo cruciale nella conformità normativa e nella protezione degli utenti.

In conclusione, il software testing è essenziale per garantire la qualità, l'affidabilità e la sicurezza del software. Attraverso il continuous testing e l'integrazione nel ciclo di sviluppo, è possibile minimizzare i rischi e soddisfare le aspettative degli utenti, contribuendo anche a costruire una solida reputazione aziendale.

Tuttavia bisogna anche dire che l'attività di testing offre tali vantaggi solamente nel momento in cui i test eseguiti danno feedback utili, fattore che potrebbe non verificarsi se la suite di test e affetta dalla cosiddetta “flakiness”.

1.1 Contesto Applicativo

Una pratica del testing ampiamente utilizzata nell'ingegneria del software è rappresentata dai test di regressione. Tali test sono essenziali per valutare se le modifiche apportate a una versione di codice esistente possono, inavvertitamente, reintrodurre errori o bug che erano stati precedentemente corretti [2]. Quando un test di regressione fallisce, ciò indica generalmente che le modifiche apportate potrebbero non essere corrette o potrebbe essere stato introdotto un nuovo bug. Tuttavia, va sottolineato che l'affidabilità dei test di regressione stessi può rappresentare una sfida. Alcuni casi di test possono, infatti, essere afflitti da un fenomeno noto come "flakiness" [3]. Questo si verifica quando un test, eseguito sulla stessa base di codice, presenta comportamenti sia di successo che di fallimento in momenti diversi, rendendolo inaffidabile e producendo risultati non deterministici [3]. Tale fenomeno può dipendere da diversi fattori, come ad esempio: concorrenza, dipendenza esterna, ritardi temporali, ordine di esecuzione etc..

La presenza di test flaky all'interno di una suite di test porta con se diversi disagi per gli sviluppatori, con conseguenze significative sia sulle fasi di sviluppo, che sulle prestazioni complessive del team, in quanto data la loro natura non deterministica è difficile eseguire il debug bloccando così il ciclo di rilascio. Inoltre un ulteriore fattore da non sottovalutare riguarda la mancanza di fiducia nei test sviluppati che si innesca nello sviluppatore, data la possibile flakiness.

Anche se minima non è rara la presenza di test flaky all'interno di un progetto, secondo la letteratura precedente sulla questione [4, 5, 6], la flakiness si verifica esplicitamente in circa il 2% dei test. Google ha riferito che il 4,56% dei fallimenti dei test sono causati da test flaky [2], e i prodotti Windows e Dynamics di Microsoft riportano un simile 5% [7]. Tuttavia è difficile stimare con precisione la quantità di test flaky all'interno di una suite di test. A causa della loro natura non deterministica, i test possono essere considerati flaky anche se non si verifica la flakiness. Questo è il motivo per cui i ricercatori hanno sostenuto la necessità di considerare tutti i test come potenzialmente flaky [8, 9].

I potenziali danni causati dalla flakiness dei test case sono stati resi sempre più popolari da professionisti e aziende di tutto il mondo (ad esempio, [5, 6]), che hanno

tutti chiesto meccanismi automatizzati per rilevarla e affrontarla.

La comunità di ricerca sull'ingegneria del software ha condotto indagini empiriche volte a definire le cause della flakiness [4, 10, 11, 3, 12], con la definizione di tecniche per rilevarle e affrontarle [13, 14, 15, 16]. Tuttavia la maggior parte delle tecniche d'identificazione richiede che i casi di test vengano eseguiti più volte: ad esempio l'approccio più noto si chiama RERUN e consiste nell'eseguire lo stesso test N volte, con N che varia da decine a migliaia di esecuzioni. Da come si può intuire tale approccio è abbastanza costoso ed inutilizzabile nella pratica; inoltre non è neanche detto che sulle N run riusciamo ad identificare la flakiness. Ad esempio è stato evidenziato come in alcuni casi rieseguendo un test flaky 10000 volte non si riesce ad identificare la sua flakiness, in quanto è necessario cambiare l'oridine d'esecuzione [17].

Data la limitazione di questo approccio, sono state sviluppate metodologie alternative come DeFlaker [13] e iDFlakies [18]. La prima di queste si concentra a livello di commit, valutando la differenza nella copertura ottenuta eseguendo lo stesso test su due commit diversi, mentre la seconda cerca di identificare la flakiness eseguendo i test con un ordine diverso. Tuttavia, è importante notare che entrambi questi approcci sono in grado di rilevare i test flaky solo dopo che sono stati introdotti nella suite di test.

Contemporaneamente, sono stati proposti diversi approcci basati sull'apprendimento automatico (machine learning) per individuare la flakiness già durante la fase di sviluppo, prima che venga introdotta nella suite di test. Pinto et al. [19] hanno presentato un approccio basato su un dizionario per identificare la potenziale presenza di flakiness all'interno di un test. Allo stesso modo, Alshammari et al. [17] hanno sfruttato metriche statiche, come le metriche della classe di produzione, i code smell e i test smell, insieme a metriche dinamiche come la copertura, per sviluppare una pipeline in grado di prevedere la flakiness con una precisione che può arrivare fino al 66%.

Recentemente, Pontillo et al. [1] hanno sviluppato una pipeline che si basa esclusivamente su metriche statiche relative alle classi di produzione e ai metodi di test, riuscendo a prevedere la flakiness con una precisione che può raggiungere il 70%.

1.2 Motivazione ed Obiettivi

Nonostante i successi ottenuti nei lavori precedenti, va sottolineato che essi hanno adottato principalmente un approccio in-vitro per affrontare il problema. Il loro obiettivo principale era valutare la fattibilità della cosa piuttosto che testarne l'applicabilità pratica in situazioni reali. In questo studio, abbiamo scelto di estendere la ricerca presentata da Pontillo et al. [1] adottando un approccio in-vivo, il che ci permetterà di valutare concretamente il suo utilizzo nell'ambiente reale.

La nostra analisi sarà condotta su due dataset diversi: FlakeFlagger e IDoFT. È importante notare che FlakeFlagger è condiviso con lo studio di Pontillo et al. [1]. La prima fase del nostro lavoro consisterà nel replicare lo studio di Pontillo et al. [1] al fine di identificare la pipeline più efficace. Successivamente, prendendo ispirazione dalle ricerche condotte nel campo della previsione dei difetti software, esamineremo la possibilità di utilizzare la pipeline identificata in ambito With-in Project e Cross-Project, pertanto parleremo di With-in Project Flakiness Prediction (WPFP) e Cross-Project Flakiness Prediction (CPFP). Questa analogia con la previsione dei difetti è possibile poiché in entrambi i contesti vengono impiegate come variabili indipendenti metriche statiche relative alle classi di produzione e ai metodi di test.

Alla luce del concetto di Cross-Project Defect Prediction (CPDP), considereremo la possibilità di predire la flakiness presente nella suite di test di un repository Y utilizzando i dati provenienti da repository simili. Inoltre, dato che l'approccio cross-project può presentare sfide legate alla differente distribuzione dei dati tra la sorgente (source) e il destinatario (target), valuteremo anche un approccio With-in Project. In quest'ultimo caso, cercheremo di prevedere la flakiness dei futuri test di un repository Y utilizzando esclusivamente i dati relativi a tale repository, superando così alcune delle sfide connesse all'approccio Cross-Project.

1.3 Risultati Ottenuti

Nel perseguire gli obiettivi stabiliti, abbiamo iniziato generando due dataset denominati "FlakeFlagger" e "IDoFT". Successivamente, è stata condotta un'attenta

fase di pulizia dei dati per migliorare la qualità delle informazioni su cui basare il nostro studio.

Come primo esperimento, seguendo il precedente lavoro di ricerca, abbiamo identificato la migliore pipeline di machine learning in grado di adattarsi ai nostri dati. Questo approccio ha non solo dimostrato la fattibilità della predizione della flakiness, ma ha anche superato le prestazioni dello stato dell'arte. Utilizzando esclusivamente metriche relative al metodo di test e alla classe di produzione, siamo riusciti a ottenere risultati promettenti. Tuttavia, benché questo approccio abbia mostrato ottime performance, è importante notare che potrebbe non essere immediatamente pratico nell'uso quotidiano.

Pertanto, abbiamo condotto ulteriori esperimenti in contesti differenti, denominati "WPFP" e "CPFP". Nel contesto "WPFP", abbiamo ottenuto risultati eccellenti, suggerendo che la previsione della flakiness mediante l'uso di un insieme di metriche statiche relative al metodo di test e alla classe di produzione potrebbe rappresentare un'alternativa valida all'approccio della "ReRuns" per l'identificazione dei test flaky.

Tuttavia, questa stessa efficienza non è stata riscontrata nel contesto "CPFP" non supervisionato, dove problemi di adattamento del dominio causati dalle metriche utilizzate hanno reso inapplicabile il nostro approccio. In contrasto, nel contesto "CPFP" supervisionato, in cui abbiamo affrontato l'adattamento del dominio utilizzando dati etichettati dalla repository di destinazione, abbiamo ottenuto prestazioni paragonabili all'approccio "WPFP".

I risultati ottenuti dimostrano che il machine learning rappresenta una possibile alternativa per la rilevazione dei test flaky. Tuttavia, va notato che al momento il suo utilizzo è limitato alle repository in cui si conosce la flakiness della suite di test della repository, rendendolo un'opzione ancora prematura per la maggior parte dei progetti in cui la flakiness rimane un argomento sconosciuto. I

1.4 Struttura della tesi

La struttura della tesi è organizzata nel seguente modo:

- **Introduzione** - Inizia introducendo il concetto di testing e il problema del-

la flakiness. Successivamente, esamina brevemente gli approcci sviluppati per affrontare questa problematica. Infine, fornisce una panoramica concisa dell'analisi condotta in questo studio e dei risultati ottenuti.

- **Background and Related Work** - Questo capitolo presenta un riassunto della letteratura esistente sulla flakiness.
- **Methodology** - Specifica in maniera dettagliata quella che è la metodologia applicata verso il problema, come è stata identificata la migliore pipeline, cosa è la valutazione within-project, cosa è la valutazione cross-project e quali tecniche sono state applicate in quest'ultimo caso.
- **Experimental Setup** - Espone inizialmente le domande di ricerca affrontate nel lavoro. Successivamente, presenta e analizza i dataset utilizzati come base per la ricerca. Infine, fornisce dettagli sulla configurazione sperimentale, garantendo la replicabilità del lavoro.
- **Risultati** - Presenta e discute i risultati ottenuti dai vari esperimenti condotti.
- **Discussioni e limiti** - Specifica le possibili minacce al lavoro e le precauzioni prese per evitarle.
- **Conclusioni** - In questo capitolo, vengono esposti i risultati dell'analisi condotta e vengono suggeriti possibili sviluppi futuri.

CAPITOLO 2

Background and Related Work

2.1 Terminologia

Test case: Secondo lo standard IEEE 829-1998 [20] un test case è definito come "un insieme di input, condizioni di esecuzione e un criterio di pass/failure". Tuttavia in accordo con i precedenti lavori [4, 3], tale definizione può essere estesa includendo fattori aggiuntivi che possono servire nel contesto della flakiness di un caso di test, come l'ambiente d'esecuzione, le dipendenze di test e il codice di produzione corrispondente.

Test di regressione: Il test di regressione è definito come "l'attività di testing che consente agli sviluppatori di controllare che le modifiche apportate al codice non abbiano introdotto difetti, che non erano presenti nel sistema" [21].

Test Flaky: Un test flaky è definito come "un test non-deterministico che mostra sia un comportamento di pass che di failure quando viene eseguito sullo stesso codice [3]. Essi comportano molti problemi durante il testing del software, come spreco di tempo e risorse computazionali, in quanto ne è difficile eseguire il debug. Molti studi sono stati condotti per studiare i test flaky, le loro cause e le soluzioni per affrontarli [22, 3, 13, 17, 19, 23]. Le cause principali includono attese asincrone, dipendenza

dall'ordine d'esecuzione, concorrenza, perdita di risorse e input e output di test errati. Tuttavia, essi sono stati associati anche ad altri fattori come test smell.

2.2 Stato dell'arte

La pratica del testing è una procedura standard utilizzata per individuare difetti all'interno del codice sviluppato. Tuttavia, un test fornisce feedback utile solo se produce risultati consistenti (pass o failure) con ogni esecuzione della stessa versione di codice. Il problema sorge quando si verificano test non deterministici, noti come "test flaky", che producono risultati pass o failure in modo intermittente, senza modifiche agli input, all'ambiente di esecuzione o al codice sottoposto a test. Questa situazione è problematica poiché crea incertezza nelle decisioni correttive [9], oltre a comportare un aumento dei costi e un ritardo nel rilascio del software.

Negli ultimi anni, la questione della flakiness dei test ha attirato l'attenzione dei ricercatori, che hanno condotto diversi studi per comprendere le cause e l'impatto dei test flaky nei software open source e proprietari. Studi sui progetti open source hanno rilevato che il 13% delle build fallite è attribuibile a test flaky [24]. Google ha segnalato che circa il 16% dei loro test è flaky, e 1 su 7 dei test scritti dai loro ingegneri occasionalmente fallisce senza alcuna modifica al codice o ai test (Micco, 2016). Nel 2020, GitHub ha riferito che un commit su undici (9%) ha causato una build rossa a causa di test flaky (Raine, 2020).

Sono state identificate diverse cause principali della flakiness, tra cui:

1. **Concorrenza**: L'esecuzione di test in parallelo o in un ambiente condiviso può causare problemi di concorrenza. Ad esempio, due test potrebbero competere per le stesse risorse o dati, causando risultati incoerenti. La gestione della concorrenza nei test è cruciale per evitare la flakiness.

2. **Dipendenza esterna**: I test che dipendono da servizi esterni, API o risorse di rete possono essere influenzati da problemi temporanei o intermittenti in tali componenti esterne. Ad esempio, un test che chiama un servizio Web potrebbe fallire se il servizio è temporaneamente indisponibile, anche se il codice del software è corretto.

3. **Stato iniziale non deterministico**: Alcuni test possono essere sensibili allo stato iniziale dell'applicazione o dell'ambiente. Se lo stato iniziale non è controllato o è influenzato da fattori esterni, come il carico del sistema, i test potrebbero produrre risultati diversi in momenti diversi.

4. **Ritardi temporali**: Alcuni test potrebbero coinvolgere ritardi temporali, come attese per il completamento di operazioni asincrone. Questi ritardi possono causare differenze nei tempi di esecuzione dei test, portando a risultati fluttuanti.

5. **Ordine di esecuzione**: L'ordine in cui vengono eseguiti i test o le operazioni all'interno di un test potrebbe influenzare i risultati. Se un test dipende dall'ordine di esecuzione di altri test o operazioni, potrebbe diventare instabile.

6. **Ambiente di esecuzione variabile**: Cambiamenti nell'ambiente di esecuzione, come configurazioni diverse o variabili d'ambiente, possono influenzare i risultati dei test. Questi cambiamenti possono essere difficili da rilevare ma devono essere considerati.

7. **Qualità dei casi di test**: I casi di test stessi potrebbero contenere difetti o essere mal progettati. Un caso di test con una logica difettosa o con aspettative poco chiare può produrre risultati inaffidabili.

Tali cause fanno si che la flakiness vada ad impattare su diversi aspetti di un sistema software. Diversi articoli citano come i test flaky possono danneggiare la build di un sistema software [25, 26], specialmente per quanto riguarda la pratica del DevOps in cui vengono utilizzate build automatizzate per garantire la cosiddetta continuous integration. Se in tali build sono poi integrate suite di test generate in maniera automatica tramite strumenti come Randoop o EvoSuite allora è ancora più probabile che esse falliscano in quanto è stato dimostrato come tali tool possono generare falicmente test flaky [27]. Automaticamente danneggiando la build viene poi rallentato quello che è il rilascio del prodotto. Tuttavia un aspetto che spesso ricade in secondo piano in quanto non legato strettamente al sistema software riguarda la produttività e la fiducia degli sviluppatore.

La presenza di test flaky induce gli sviluppatori a predere fiducia nell'utilità della suite di test in generale (Palmer, 2019) e a perdere fiducia nella loro build. (Grant, 2016). Inoltre i test flaky possono innescare anche un "danno collaterale" per gli sviluppatori: se lasciati incontrollati o irrisolti, possono avere un impatto maggiore e possono rovinare il valore di un'intera suite di test (Lee, 2020).

Il vero costo flakinees è una mancanza di fiducia nei tuoi test..... Se non hai fiducia nei tuoi test, allora non sei in una posizione migliore di un team che ha zero test. I test flaky avranno un impatto significativo sulla tua capacità di consegnare con sicurezza e continuita. (Spotify Engineering, Palmer (2019)).

Dato quindi il forte impatto della flakiness, nel corso degli anni sono state sviluppate diverse tecniche per quanto riguarda la detection. Esse possono essere raggruppate in due categorie di approccio differente, ovvero, tecniche dinamiche che richedono l'esecuzione dei test e tecniche statiche che analizzano il codice di test e la relativa classe di produzione.

Per quanto riguarda le tecniche dinamiche diciamo che l'approccio più semplice per l'identificazione dei test flaky e quello della ReRuns, ovvero rieseguire più volte una suite di test ed osservare il comportamento dei singoli test. Lam et al. [18] hanno migliorato tale approccio introducendo iDFlakies, uno strumento che rileva test flaky eseguendo nuovamente i test in diversi ordini. Tuttavia questa strategia può risultare fattibile nel momento in cui la suite di test è piccola e la sua esecuzione non ha costi elevati. Se pensiamo a grandi progetti software i quali possiedono una grande suite di test tale approccio risulta essere impraticabile dati gli elevati costi. Tali costi sono poi accentuati ancora di più visto che ad oggi non esiste un numero N di esecuzioni che ci garantisce che un determianto test sia flaky o meno, pertanto maggiori sarranno le esecuzioni maggiore sarà la probabilità che quel test non sia flaky. Visti i limiti di tale approccio Bell et al. [13] hanno proposto una tecnica di detection dinamica differente, DeFlaker. Esso è uno strumento che analizza le differenze nella code coverage tra un commit e l'altro per avvisare gli sviluppatori dell'emergere di una sorta di flakiness. Anche se DeFlaker può essere vista come una possibile alternativa a basso costo per la detection come iDFlakies per progettazione è in grado di rilevare la flakiness solo

dopo la sua comparsa, vale a dire solo dopo che gli sviluppatori hanno introdotto test flaky. In questo senso, potrebbero essere utili per diagnosticare test flaky, ma non per impedire la loro introduzione.

Pertanto la ricerca si è spostata sulle tecniche statiche, che sfruttando il machine learning riescono sia a ridurre i costi che ad identificare la flakiness in maniera preventiva. Pinto et al. [19] hanno presentato un approccio basato su un dizionario per identificare la potenziale presenza di flakiness all'interno di un test. Allo stesso modo, Alshammari et al. [17] hanno sfruttato metriche statiche, come le metriche della classe di produzione, i code smell e i test smell, insieme a metriche dinamiche come la copertura, per sviluppare una pipeline in grado di prevedere la flakiness con una precisione che può arrivare fino al 66%. Recentemente, Pontillo et al. [1] hanno sviluppato una pipeline che si basa esclusivamente su metriche statiche relative alle classi di produzione e ai metodi di test, riuscendo a prevedere la flakiness con una precisione che può raggiungere il 70%. Tuttavia bisogna dire che tali lavori si sono soffermati solamente sulla fattibilità della cosa, senza dimostrare la possibilità di un reale utilizzo e quindi la sua efficacia pratica. Pertanto ad oggi la flakiness risulta essere ancora un problema persistente nello sviluppo dei sistema software il quale spesso viene trascurato dati gli elevati costi per l'identificazione dei test flaky e il debugging.

CAPITOLO 3

Metodologia

I lavori precedenti nella letteratura presentano una caratteristica comune: si concentrano sull'identificazione di esempi di test flaky e non flaky al fine di creare un dataset in modo da costruire una pipeline per la predizione della flakiness. Tale pipeline viene poi validata su un sottoinsieme dell'intero dataset, detto test-set. Tuttavia, è importante sottolineare che questo approccio nel momento in cui le predizioni si basano solamente sull'utilizzo di metriche statiche relative al codice sorgente, sembra dimostrare principalmente la fattibilità del predirre la flakiness di un test, piuttosto che un reale utilizzo.

Il nostro studio si è proposto di approfondire ulteriormente questi lavori, con un'attenzione particolare al lavoro di Pontillo et al. [1], con l'obiettivo di valutare se la predizione della flakiness tramite machine learning potesse rappresentare un'alternativa valida all'attuale tecnica più diffusa per l'identificazione dei test flaky, ovvero la RERUN.

Nella fase iniziale dello studio, abbiamo replicato il lavoro di Pontillo et al. [1] creando un dataset su cui poi è stata identificata la miglior pipeline. Essa è stata costruita combinando diverse operazioni di pre-processing:

- **Feature Scale**:

  - Normalizzazione: La normalizzazione Min-Max è una tecnica di trasformazione dei dati utilizzata per scalare i valori di una variabile in un intervallo specifico, di solito tra 0 e 1.
  - Standardizzazione: La standardizzazione Z-Score è una tecnica statistica utilizzata per trasformare una distribuzione di dati in modo da avere una media (valore atteso) di zero e una deviazione standard di uno.

- **Feature Selection (Multicollinearity & Information Gain)**: La multicollinearità è un fenomeno che si verifica quando due o più variabili indipendenti in un modello statistico o di regressione sono altamente correlate tra loro e l'Information Gain è una misura che aiuta a identificare le variabili più informative o rilevanti per la classificazione o la previsione

- **Feature Construct (Principal Component Analysis)**: La Principal Component Analysis (PCA) è una tecnica di riduzione della dimensionalità utilizzata nell'ambito dell'analisi dei dati e dell'apprendimento automatico. L'obiettivo principale della PCA è quello di trasformare un insieme di dati complesso, spesso caratterizzato da molte variabili correlate tra loro, in un nuovo insieme di variabili non correlate chiamate "componenti principali". Queste componenti principali sono ordinate in modo che la prima contenga la massima varianza nei dati, la seconda la seconda massima varianza, e così via.

- **Data Balancing (SMOTE)**: SMOTE è una tecnica efficace per affrontare il problema dello sbilanciamento di classi e può migliorare significativamente le prestazioni dei modelli di machine learning quando si tratta di dataset con classi fortemente sbilanciate.

Iniziando da una pipeline di base senza alcuna operazione di pre-processing sono state valutate ben 18 diverse pipeline in combinazione con diversi classificatori, ossia: *KNN, SVM, Logistic Regression, Decision Tree, Random Forest e XGBClassifier*.

1. *Pipeline senza pre processing*
2. *Pipeline con Standardizzazione*
3. *Pipeline con Normalizzazione*

4. *Pipeline con Multicollinearity & IG*
5. *Pipeline con PCA*
6. *Pipeline con SMOTE*
7. *Pipeline con Standardizzazione e Multicollinearity & IG*
8. *Pipeline con Standardizzazione e PCA*
9. *Pipeline con Standardizzazione e SMOTE*
10. *Pipeline con Standardizzazione, Multicollinearity & IG e SMOTE*
11. *Pipeline con Standardizzazione, PCA e SMOTE*
12. *Pipeline con Normalizzazione e Multicollinearity & IG*
13. *Pipeline con Normalizzazione e PCA*
14. *Pipeline con Normalizzazione e SMOTE*
15. *Pipeline con Normalizzazione, Multicollinearity & IG e SMOTE*
16. *Pipeline con Normalizzazione, PCA e SMOTE*
17. *Pipeline con PCA e SMOTE*
18. *Pipeline con Multicollinearity & IG e SMOTE*

Come metrica di valutazione, abbiamo utilizzato Precision, Recall e F1-Score trascurando invece l'Accuracy. Tale decision è stata presa poichè essendo il problema di classificazione fortemente sbilanciato, affidarsi all'accuracy non è un ottima scelta in quanto quest'ultima può risultare alta anche senza individiuare nessun test flaky.

$$F1 - Score = \frac{2 * Recall * Precision}{Recall + Precision}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

Dopo aver individuato la miglior pipeline, siamo passati ad eseguire un ulteriore validazione attraverso due approcci distinti: inizialmente, abbiamo adottato un approccio "Within-Project Flakiness Prediction (WPFP)", seguito poi da un approccio "Cross-Project Flakiness Prediction (CPFP)".

Nel primo caso, abbiamo simulato un ambiente in cui si prevede la flakiness dei metodi di test all'interno di una repository utilizzando esclusivamente le informazioni relative a quella specifica repository. Questo ci ha consentito di dimostrare che, lavorando all'interno di una repository di cui conosciamo i metodi di test affetti da flakiness e non, utilizzando esclusivamente le metriche statiche proposte da Pontillo et al. [1], è possibile prevedere la flakiness di futuri metodi di test.

Tuttavia, prevedere la flakiness in un contesto WPFP, anche se possibile, presenta un notevole limite. È necessario disporre di informazioni sufficienti sulla suite di test della repository, poiché durante l'addestramento della pipeline vengono utilizzate esclusivamente queste informazioni. Questa limitazione non è presente in un contesto CPFP, perchè le tecniche lavorano in modo da utilizzare le informazioni relative ad altre repository per predire la flakiness in una repository target.

Inizialmente la valutazion CPFP è stata condotta estraendo in maniera iterativa una repository (target) alla volta dal dataset general, per poi addestrare la pipeline su quest'ultimo e predirre sul target. Tuttavia, prendendo spunto da quella che è la defect prediction, ovvero il contesto in cui tramite tecniche di ML si cercano di predirre difetto o errori nel codice sorgente prima che causino problema, sono state esplorate ulteriori tecniche Cross-Project da combinare con la pipeline individuata. Il tutto è stato possibile in quanto sia nella defect prediction che in tale lavoro vengono utilizzate metriche statiche riguardanti il codice sorgente. Di cui molte risultano essere in comune.

3.1 Metodi Cross Project Flakiness Prediction

La Cross-Project Predict si basa su quello che è il transfer learning, ovvero: dato un dominio sorgente D_S, un task T_S, un dominio target D_T e un task target T_T, il transfer learning mira a migliora l'apprendimento della funzione predittiva f_T in D_T utilizzando le conoscenze in D_S e T_S dove $D_S \neq D_T$ o $T_S \neq T_T$.

Il transfer learning può essere implementato in diverse modalità, ognuna delle quali affronta il problema in modi distinti:

1. **Feature-Based**: Questo metodo si basa sull'identificazione di caratteristiche comuni che mostrano comportamenti simili rispetto al compito tra il dominio sorgente e quello target. In questa prospettiva, si crea una nuova rappresentazione delle caratteristiche (encoded feature space) che mira a mitigare le differenze tra le distribuzioni dei dati tra i due domini. Successivamente, il compito viene appreso nel encoded feature space.

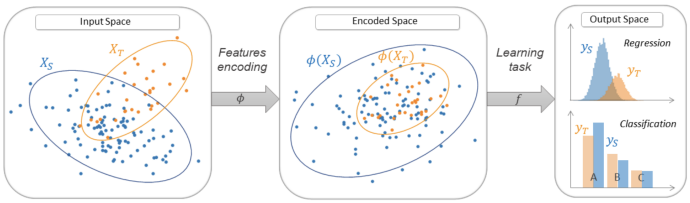

Figura 3.1: TL Feature-Based

2. **Instance-Based**: Qui, per affrontare le differenze tra le distribuzioni dei dati tra il dominio sorgente e quello target, si procede con la ridefinizione dei pesi dei dati di addestramento. Questo significa che si assegna maggiore importanza o peso ai dati del dominio sorgente per adattare il modello al dominio target.

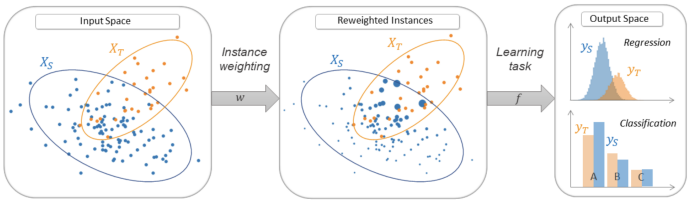

Figura 3.2: TL Instance-Based

3. **Parameter-Based**: In questo caso, i parametri di un modello precedentemente addestrato sul dominio sorgente vengono adattati o modificati in modo da renderli applicabili ed efficaci sul dominio target.

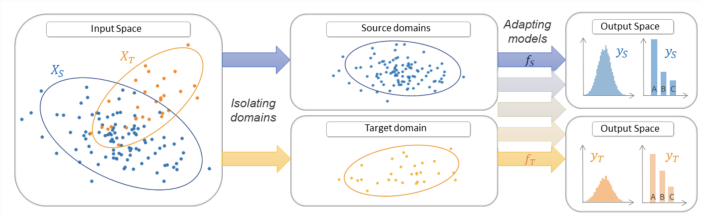

Figura 3.3: TL Parameter-Based

Una caratteristica cruciale delle tre metodologie di transfer learning sopra citate riguarda la quantità di dati etichettati richiesti nel dominio target per applicarle con successo. Nel metodo basato sulle feature (Feature-Based), non è necessario disporre di dati etichettati nel dominio target per iniziare il processo di trasferimento, mentre per le altre metodologie è necessaria una quantità significativa di dati etichettati del dominio target. Ad esempio, per pesare correttamente le istanze o adattare i parametri in modo adeguato al dominio target, dati etichettati specifici di quest'ultimo sono indispensabili.

In tale studio sono state esplorate le prime due metodologie, al fine di valutarne l'efficacia della CPFP in maniera supervisionata e non.

3.1.1 Feature-Based Transfer Learning

Filtro di Burak

La prima metodologia feature-based provata è stato il filtro di burak. Turhan et al. [28] hanno dimostrato che l'utilizzo di tutti i dati di un dataset aggregato per addestrare modelli predittivi per la defect prediction porta a un numero eccessivo di falsi allarmi. Pertanto, hanno proposto una tecnica di filtraggio che guida la selezione dei dati di addestramento utilizzando i dati di test, in cui si selezionano le k istanze di training più vicine, calcolate con distanza euclidea, per ogni istanza di test. Nel nostro studio, abbiamo applicato il filtro Burak con la stessa configurazione utilizzata da Turhan et al. [28] che ha selezionato un valore di k = 10.

Local Model

Menzies et al. [29, 30] hanno proposto modelli locali, che comprendono il raggruppamento dei dati di training e l'addestramento di modelli diversi, uno per ogni

cluster, utilizzando solo i dati del cluster. Quando è necessario classificare nuovi dati, le istanze vengono assegnate al cluster più vicino e quindi classificate utilizzando il modello corrispondente. I loro studi empirici mostrano che i modelli addestrati su dati locali ottengono risultati migliori di quelli addestrati su tutti i dati. Sono stati provati due approcci differenti per quanto riguarda i modelli locali, uno basato sul clustering ed il secondo basato sulle repository. Nel primo metodo utilizzando l'algoritmo di clustering K-Means vengono generati i vari cluster a partire dai dati di training, per poi addestrare una pipeline per ogni cluster. Per classificare nuovi dati tramite il K-Means viene identificato prima il cluster d'appartenenza X e successivamente la predizione viene eseguita utilizzando la pipeline addestrata sul cluster X.

Per quanto riguarda i modelli locali basati su repository, i dati di training vengono raggruppati in base alla repository d'appartenenza. Per ogni repository viene calcolato prima il centroide che corrisponde al vettore $[m_1, m_2, ..., m_N]$ dove m_i corrisponde alla media della feature i e poi viene addestrata una propria pipeline. Nel momento in cui si vuole classificare una nuova istanza, tramite la distanza euclidea si identifica il centroide più vicino ed in base al centroide ottenuto si utilizza la pipeline della relativa repository per eseguire la predizione.

TCA : Transfer Component Analysis

Transfer Component Analysis (TCA) [31], è una strategia di adattamento di dominio che si propone di apprendere un sottospazio condiviso tra un dominio sorgente e uno target. Questo sottospazio condiviso è composto da componenti di trasferimento appresi all'interno di uno spazio di Hilbert a kernel riproducente (RKHS) [32] utilizzando la massima discrepanza media (MMD) [33]. Nell'ambito di questo sottospazio, l'obiettivo è far sì che le distribuzioni dei dati provenienti da domini diversi convergano e, allo stesso tempo, preservare le informazioni cruciali relative alle caratteristiche originali dei dati.

Da notare che la TCA ha dimostrato di ottenere notevoli successi nel contesto della defect prediction, e, in effetti, è considerata una delle migliori opzioni disponibili nella letteratura per approcci cross-project. Tuttavia, è importante sottolineare che questa

tecnica è afflitta da una complessità computazionale considerevole, che è strettamente legata alle dimensioni del dominio di origine e del dominio di destinazione.

Per mitigare questa problematica, nel presente lavoro abbiamo introdotto una fase di undersamplig che precede l'applicazione della TCA riducendo la size del source set ad un rapporto di 3:1 ovvero ogni test flaky saranno mantenuti 3 test non flaky.

IG_SM_FS_TCA

Prendendo ispirazione dal lavoro di Khatri et al. [34], è stata sviluppata una strategia di adattamento del dominio denominata IG_SM_FS_TCA. Questa strategia è caratterizzata da una sequenza di passaggi che prevedono una riduzione iniziale del numero di feature tra il dominio sorgente e quello target, seguita dall'applicazione della Transfer Component Analysis (TCA) con il realtivo pre-processing. Nel contesto della selezione delle feature più rilevanti tra il dominio sorgente e quello target, il processo si articola in diverse fasi, ovvero Feature Importances, Feature Distribution Similarity e Feature Weight. Questi passaggi ci consentono di individuare le feature più cruciali per entrambi i domini, ottimizzando così l'adattamento.

1. **Feature Importances**: Utilizzando l'Information Gain (IG), calcoliamo l'importanza delle feature nel dominio sorgente. Tuttavia, se ci fermassimo a questa fase e selezionassimo solo le K feature più importanti, queste potrebbero essere rilevanti solo per il dominio sorgente e non necessariamente per il dominio target. Pertanto, calcoliamo quindi anche la "distribution similarity" (similarità di distribuzione).

2. **Feature Distribution Similarity**: La similarità di distribuzione viene valutata utilizzando 10 misure statistiche (minimo, massimo, range, interquartile range, media, mediana, varianza, deviazione standard, skewness e kurtosis). Queste misure vengono utlizzate per calcolare il vettore delle distribuzioni S_i e T_i per ogni feature F_i^S e F_i^T con $(1 \leq i \leq K)$ nel dominio sorgente e nel dominio target.

$$S_i = [sv_{i1}, sv_{i2}...sv_{i10}]$$

$$T_i = [tv_{i1}, tv_{i2}...tv_{i10}]$$

Ogni coppia $(sv_{ij}, tv_{ij}), (1 \leq j \leq 10)$, rappresenta una misura statistica calcolata per la stessa i-esima feature sul dominio sorgente e target.

La feature Distribution Similarity della i-esima feature $FDS(F_i^S, F_i^t)$ corrisponde:

$$FDS(F_i^S, F_i^t) = \frac{\sum_{j=1}^{10} m(sv_{ij}, tv_{ij})}{10}$$

Dove:

$$m(sv_{ij}, tv_{ij}) = \frac{sv_{ij}}{tv_{ij}}, \quad if \quad sv_{ij} \leq tv_{ij}, \quad \frac{tv_{ij}}{sv_{ij}}, \quad esle$$

3. **Feature Weight**: Sfruttando l'importanza delle feature calcolata tramite IG e la FDS, pesiamo le singole feature. In questo modo, selezionando le K feature con il peso più elevato, otteniamo un insieme di feature rilevanti sia per il dominio sorgente che per quello target. La funzione di pesatura di una features F_i è definita come segue:

$$F_i(W) = \alpha * IG(F_i^S) + (1 - \alpha) * FDS(F_i^S, F_i^t)$$

Questa strategia complessa e articolata ci consente di selezionare in modo accurato le feature più importanti e di massimizzare l'adattamento tra il dominio sorgente e quello target, contribuendo così a migliorare le prestazioni complessive dell'apprendimento automatico in scenari di adattamento di dominio.

3.1.2 Instance-Based Transfer Learning (TrAdaBoost)

AdaBoost è un metodo tradizionale di apprendimento automatico [35]. La sua idea principale è quella di formare un insieme di weak learners attraverso un processo di iterazione. In ogni iterazione, uno weak learner viene ottimizzato in base alle istanze che sono state erroneamente classificate nell'iterazione precedente. Il peso di ogni istanza viene poi aggiornato in base al risultato della classificazione nell'iterazione precedente. Se un'istanza è erroneamente classificata nell'iterazione precedente, allora il suo peso è aumentato nell'iterazione corrente. Inoltre, ogni iterazione produce uno weak learner che ha anche un peso calcolato in base al tasso di errore del risultato dell'iterazione. Il suo risultato finale è una somma ponderata di tutti i weak learner. TrAdaBoost [36] è un'estensione di AdaBoost. Il suo set di

addestramento è composto da due parti: istanze di addestramento da un dominio sorgente e istanze di addestramento da un dominio target. La sua idea di base è quella di filtrare quelle istanze nel dominio sorgente, che sono dissimili da quelle nel dominio target. In TrAdaBoost, AdaBoost viene ancora utilizzato per le istanze nel dominio target durante il processo di potenziamento. Tuttavia, per le istanze nel dominio sorgente, se sono classificate erroneamente, i loro pesi sono diminuiti secondo Hedge(β) [35] al fine di ridurre al minimo le loro influenze. Soprattutto, se un'istanza x nel dominio sorgente è classificata erroneamente, il suo peso viene moltiplicato per $\beta^{|h_t(x)-c(x)|\in(0,1]}$, dove $h_t(x)$ è il risultato previsto di x da parte del weak learner e c(x) è un numero reale che rappresenta l'etichetta di x.

CAPITOLO 4

Experimental Setup

In questa sezione, esplicitiamo le domande di ricerca che costituiscono il fulcro del nostro studio. Successivamente, forniamo una panoramica dettagliata dei dataset utilizzati in questo contesto di ricerca, inclusi i loro attributi e le caratteristiche chiave. In seguito, esaminiamo i dettagli dell'esperimento che è stato condotto per rispondere alle nostre domande di ricerca, delineando le procedure, le condizioni e le metriche valutative impiegate. Infine, procediamo con l'analisi dei risultati emersi dall'esperimento condotto.

4.1 Research Questions

Per dimostrare che la detection della flakiness tramite machine learning, utilizzando solamente delle metriche statiche risulta essere una valida alternativa all'attuale tecnica più diffusa ovvero la RERUN, sono stati condotti diversi esperimenti per rispondere alle seguenti domande di ricerca:

RQ1 Quanto è efficace un approccio basato sul Machine Learning per il rilevamento della flakiness, utilizzando la classica valutazione train-validation-test?

RQ2 Quanto è efficace un approccio basato sul Machine Learning per il rilevamento della flakiness in una validazione within-project?

RQ3 Quanto è efficace un approccio basato sul Machine Learning per il rilevamento della flakiness in una validazione cross-project?

4.2 Dataset

Per costruire un dataset che affronti il problema della flakiness dei test, è essenziale raccogliere esempi di test sia flaky che non flaky al fine di analizzarne le caratteristiche distinte. In questo contesto, nonostante sia una sfida ottenere esempi di test flaky, nel nostro studio abbiamo creato due dataset: *FlakeFlagger* e *IDoFT*, il primo dei quali condiviso con il lavoro di Pontillo et al. [1].

Il dataset "FlakeFlagger" è stato creato basandosi sul lavoro di Alshammari et al. [17], i quali hanno eseguito la suite di test di 24 progetti Java open source per un totale di 10.000 esecuzioni al fine di classificare i test come flaky o non flaky. La lista completa dei test di *FlakeFlagger* è disponibile qui.

Per quanto riguarda il secondo dataset, *IDoFT*, abbiamo sfruttato le informazioni fornite dal International Dataset of Flaky Test (IDoFT). Questo dataset contiene una collezione di test flaky e fornisce dettagli come la repository GitHub in cui è stato individuato il test flaky, il commit associato, il modulo contenente il test flaky e il nome del test stesso. Tuttavia, a differenza del primo dataset, non si dispone di informazioni relative al numero di esecuzioni delle suite di test, il che introduce un potenziale bias nel modello finale. Si può trovare la lista completa dei test flaky dell'International Dataset of Flaky Test qui.

Dopo aver identificato i test flaky per costruire i dataset su cui eseguire il nostro studio, abbiamo definito le variabili dipendenti e indipendenti da utilizzare nella previsione della flakiness. La variabile dipendente principale è "isFlaky", che assume il valore 0 per i test non flaky e 1 per quelli flaky.

Per quanto riguarda le variabili indipendenti, dovendo approfondire il lavoro di Pontillo et Al. [1] abbiamo seguito il loro stesso set di variabili. Questo set comprende un totale di 25 features, che possono essere categorizzate in tre gruppi: metriche

relative al codice di produzione e ai test, code smells e test smells. Un elenco completo di queste features è riportato nella Tabella 4.1. Nel dettaglio, Pontillo et al. [1] hanno utilizzato 10 features relative al codice di produzione e ai test per valutare la dimensione e la complessità del codice di produzione e/o dei test stessi. La scelta di queste metriche è stata basata sull'ipotesi di determinare se dimensioni e complessità elevate del codice impattino sulla flakiness dei test. Inoltre, hanno incluso 5 code smells e 8 test smells, tra cui gli ultimi sono stati identificati come correlati alla flakiness dei test in uno studio precedente condotto da Camara et al. [**?**].

L'estrazione di queste metriche è stata eseguita utilizzando lo stesso strumento impiegato da Pontillo et al. [1], il quale è disponibile come appendice online al loro lavoro [1]. Per quanto riguarda le metriche relative al codice di produzione e ai test, Pontillo et al. [1] hanno sviluppato uno strumento specifico nel loro laboratorio, mentre per l'individuazione dei code smells hanno utilizzato lo strumento Decor, e per l'individuazione dei test smells hanno impiegato VITRuM.

È importante sottolineare che lo strumento utilizzato presenta alcune limitazioni legate all'associazione tra i casi di test e il corrispondente codice di produzione. Pontillo et al. [1] hanno adottato un approccio basato sulle convenzioni di denominazione, che consiste nell'utilizzare il nome di una classe di produzione (ad esempio, "ClassName") come base per identificare la classe di test corrispondente, la cui denominazione include il prefisso/postfisso "Test" (ad esempio, "TestClassName" o "ClassNameTest"). Quando questo schema di corrispondenza non era soddisfatto, la classe di produzione associata al caso di test non poteva essere individuata e, di conseguenza, il test veniva escluso dall'analisi. Tuttavia, questa limitazione era già nota nel precedente lavoro, ma veniva accettata in quanto l'approccio basato sulle convenzioni di denominazione poteva essere applicato su larga scala in modo più efficiente rispetto ad approcci più complessi basati sullo slicing statico e dinamico.

In questo studio successivo, lo strumento utilizzato da Pontillo et al. [1] è stato ulteriormente migliorato. È stato creato uno strumento unificato che, a partire da una lista di test flaky e non flaky che include i seguenti campi: URL del progetto, SHA del commit della repository , nome completo del test (packageName.ClassName.methodName) e un indicatore "isFlaky" (questo campo è necessario solo se si dispone di una lista mista di test flaky e non flaky; altrimenti, tutti i test

elencati vengono considerati flaky), esegue automaticamente il clone dei repository, l'estrazione delle metriche e la generazione del dataset. Inoltre, per garantire la corretta esecuzione dello strumento esso è stato dockerizzato. È possibile trovare il tool sulla repository GitHub associata a questo studio, disponibile al seguente link: Flakiness Detection: An Extensive Analysis

Tabella 4.1: Variabili Indipendenti

Nome	Descrizione	Calcolato su...
Metriche di qualità		
CBO	Accoppiamento tra oggetto, cioè il numero di dipendenze che una classe ha con un'altra classe	Classe di produzione
Halstead Length	Il numero totale di occorrenze dell'operatore e il numero totale di occorrenze dell'operando	Classe di produzione
Halstead Vocabulary	Il numero totale di operatori e operandi distinti in una funzione	Classe di produzione
Halstead Volume	Rappresenta la dimensione, in bit, dello spazio necessario per memorizzare il programma	Classe di produzione
LOC	Linee di codice, comprese anche quelle commentate	Classe di produzione
LCOM2	Percentuale di metodi che non accedono a un attributo specifico in media su tutti gli attributi della classe	Classe di produzione
LCOM5	Densità degli accessi agli attributi per metodo	Classe di produzione
McCabe	Numero di percorsi linearmente indipendenti attraverso il codice sorgente di un programma	Test Method
MPC	Misura il numero di messaggi che passano tra gli oggetti della classe	Classe di produzione
RFC	Numero di metodi (compresi quelli ereditati) che possono potenzialmente essere chiamati da altre classi	Classe di produzione

TLOC	Numero di linee di codice della classe di test	Test Method
WMC	Somma della complessità (cioè la complessita ciclomatica di McCabe) di tutti i metodi in una classe	Classe di produzione
Code Smells		
Class Data Should Be Private	Quando una classe espone i suoi attributi, violando il principio di information hiding	Classe di produzione
Complex Class	Quando una classe ha una complessità ciclomatica alta	Classe di produzione
Functional Decomposition	Quando in una classe ereditarietà e polimorfismo sono usati male	Classe di produzione
God Class	Quando una classe ha una dimensione elevata poichè implementa diverse funzionalità	Classe di produzione
Spaghetti Code	Quando una classe non ha struttura e dichiara un metodo lungo senza parametri	Classe di produzione
Test Smells		
Assertion Density	Il metodo di test ha un numero elevato di asserzioni	Test Method
Assertion Roulette	Quando un metodo di test ha diverse asserzioni non documentate	Test Method
Conditional Test Logic	La logica condizionale all'interno di un test diventa complessa o difficilmente comprensibile.	Test Method
Eager Test	Il metodo di test invoca diversi metodi della classe di produzione	Test Method
Fire and Forget	Un test che è a rischio di terminare prematuramente perché non attende correttamente i risultati delle chiamate esterne	Test Method
Mystery Guest	Il metodo di test utilizza risorse esterne come file o database	Test Method

Resource Optimism	Quando un metodo di test fa un'assunzione ottimistica che la risorsa esterna (ad esempio, File), utilizzata esista	Test Method
Sensitive Equal	Quando il metodo toString viene utilizzato all'interno di un metodo di test	Test Method

Nella Figura 4.1, sono riportate le proporzioni dei due dataset creati; da come si può vedere entrambi hanno uno sbilanciamento verso i test non flaky. Questa è una caratteristica che rispecchia la realtà in quanto secondo diversi studi la flakiness si verifica esplicitamente in cira il 2% dei test.

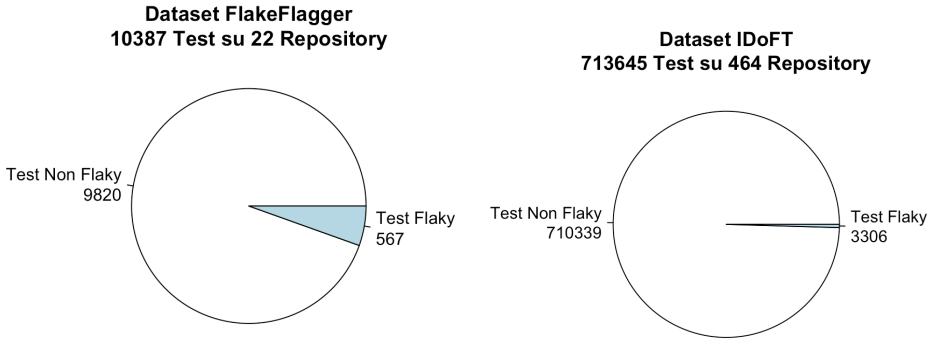

Figura 4.1: Dataset FlakeFlagger & IDoFT

4.2.1 Data Analysis

Dopo la generazione dei dataset, è stata condotta una dettagliata sessione di analisi dei dati. Questa fase è iniziata con un rigoroso processo di pulizia dei dati, mirato a migliorare la qualità dei dataset, prima di procedere con un'analisi statistica approfondita.

Per quanto riguarda la fase di pulizia dei dati, sono stati applicati una serie di filtri mirati per garantire che i dati fossero affidabili e pronti per l'analisi. Questi filtri includono:

1. **Eliminazione delle repository senza test flaky**: È stato osservato che il metric extractor utilizzato per estrarre le variabili indipendenti potrebbe non

individuare correttamente la classe di produzione di una classe di test. Di conseguenza, alcuni testo potrebbero essere ignorati, portando al rischio di avere repository nel dataset da cui non è stato estratto alcun test flaky.

2. **Verifica della presenza di repository con più commit**: All'interno dell'International Dataset of Flaky Test (IDoFT), sono presenti repository in cui i test flaky sono distribuiti su più commit diversi. Per evitare duplicazioni e semplificare l'analisi, si è scelto di mantenere solo il commit con il maggior numero di test flaky associati.

3. **Rimozione di test setup e teardown**: I test di setup e teardown sono comunemente utilizzati per configurare e ripulire l'ambiente di test prima e dopo l'esecuzione dei test principali. Tuttavia, per focalizzare l'attenzione esclusivamente sui test flaky che influenzano la qualità del software, questi test sono stati esclusi dalla nostra analisi.

4. **Rimozione dei duplicati**: I test duplicati possono rappresentare uno spreco di risorse e potrebbero influenzare negativamente i risultati dell'analisi. Pertanto, procederemo con la rimozione di eventuali test duplicati generati dal metric extractor per estrarre le metriche.

5. **Rimozione rumore**: All'interno del dataset ci potrebbero essere test etichettati come falky e test etichettati come non falky che hanno le stesse varaibili dipendenti. Questa sitazione si verifica in quanto per quei test etichettati come non flaky non è stata ancora rilevata la loro flakiness. In tal caso per evitare di avere del rumore all'interno dei dati tali test saranno rimossi.

Come si può osservare dalla Tabella 4.2, l'applicazione dei filtri ha comportato una notevole riduzione del numero di test in entrambi i dataset. Questa diminuzione è stata particolarmente evidente nel caso del dataset "IDoFT", dove il numero di test è sceso da 713.645 a 277.690. Le ragioni di questa riduzione possono essere ricondotte principalmente a due fattori chiave.

In primo luogo, si è verificata una significativa diminuzione dovuta alla presenza di numerose repository (un totale di 119) per le quali non è stato possibile estrarre alcun test flaky.

Tabella 4.2: Data Cleaning

Filtro	FlakeFlagger			IDoFT		
	TF	TNF	Repo	TF	TNF	Repo
Senza Filtro	567	9820	22	3306	710339	464
Filtro 1	567	9262	19	3306	621335	345
Filtro 2	567	9262	19	2853	314915	254
Filtro 3	560	9173	19	2804	295369	254
Filtro 4	559	9068	19	2790	275383	254
Filtro 5	559	8925	19	2790	274900	254

In secondo luogo, l'eliminazione delle repository con più commit ha ulteriormente contribuito a questa riduzione, con un totale di 100 repository duplicate rimosse.

Dopo la fase di pulizia dei dati, è stata condotta un'analisi statistica dei due dataset. Inizialmente, è stata esaminata la composizione dei dataset, con un focus sulla distribuzione delle repository in base ai loro test flaky (come mostrato in Figura 4.2). Successivamente, è stata esaminata la bilanciatura delle varie repository (come evidenziato nelle Figure 4.3 e 4.4).

Per quanto riguarda la Figura 4.2, è evidente come nel dataset "FlakeFlagger" la distribuzione delle repository in relazione ai propri test flaky sia molto uniforme. Al contrario, nel dataset "IDoFT", ben 201 delle 254 repository possiedono meno di 10 test flaky, il che spiega il forte sbilanciamento verso i test non flaky rispetto al dataset "FlakeFlagger". In IDoFT, solo l'1% dei test sono flaky, mentre in FlakeFlagger il 5,9% dei test presentano questa caratteristica.

Le Figure 4.3 e 4.4 ci mostra che in entrambi i dataset la maggioranza delle repository è bilanciata a favore dei test non flaky, riflettendo così il contesto reale. Solo una repository per "FlakeFlagger" e sei per "IDoFT" presentano uno sbilanciamento a favore dei test flaky.

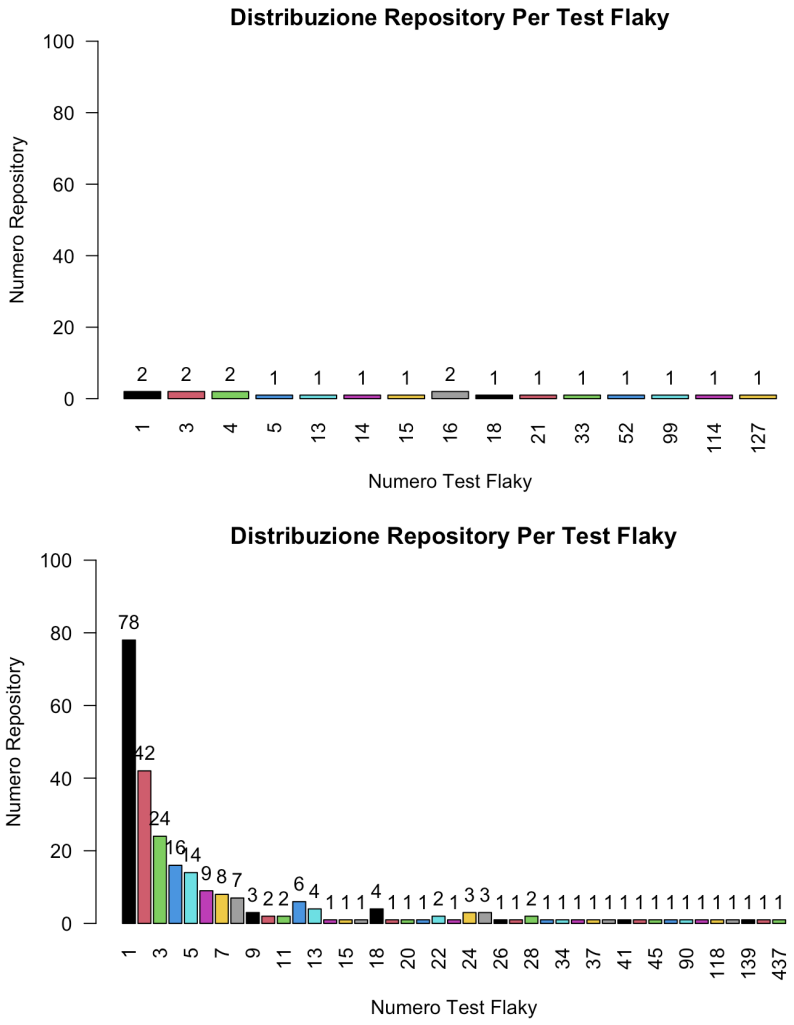

Figura 4.2: Distribuzione Test Flaky Repository FlakeFlagger & IDoFT

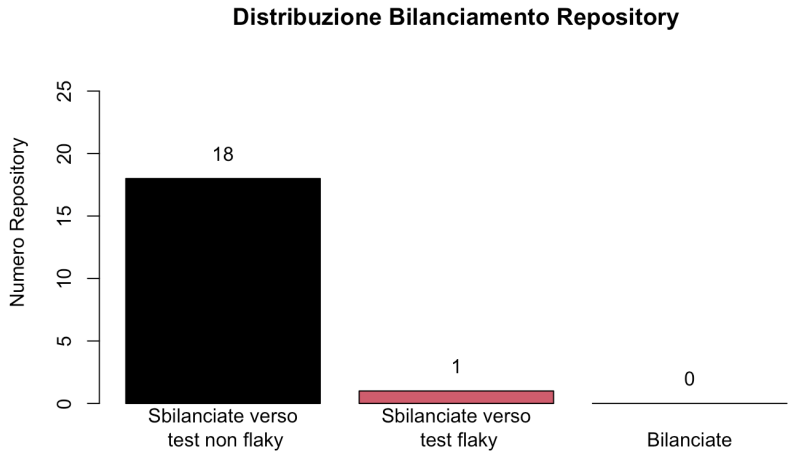

Figura 4.3: Distribuzione Bilanciamento Repository FlakeFlagger

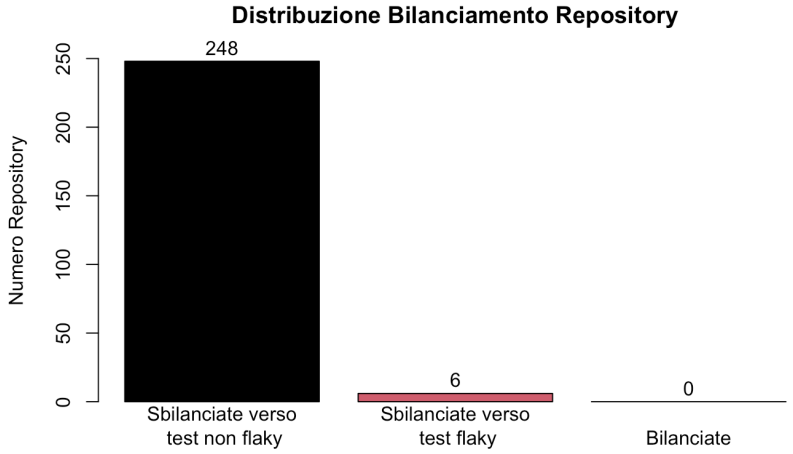

Figura 4.4: Distribuzione Bilanciamento Repository IDoFT

4.3 Experimental Design

L'esperimento è iniziato con la progettazione di una pipeline per la previsione della flakiness dei test. Per raggiungere questo obiettivo, si è proceduto alla suddivisione del dataset in tre parti: train (60%), validation (20%) e test (20%). Il validation set è stato utilizzato per valutare diverse configurazioni della pipeline al fine di individuare la soluzione ottimale, la quale è stata successivamente validata sul set di test.

Dopo aver convalidato la miglior pipeline, si è ritenuto opportuno condurre un ulteriore test per effettuare un confronto diretto con lo stato dell'arte utilizzando il dataset *FlakeFlagger*. Come nei lavori di ricerca precedenti, la miglior pipeline è stata sottoposta a una ulteriore valutazione mediante una procedura di cross-validation stratificata a 10 fold. Questo approccio garantisce che ogni caso di test presente nel dataset appaia almeno una volta nel set di test. Per ciascuna repository, sono poi stati calcolati i seguenti indicatori: *True Positive*, *False Negative*, *False Positive*, *True Negative*, *Precision*, *Recall* e *F1-Score*.

Tuttavia, va sottolineato che questo studio, non vuole essere una replica dei precedenti lavori allo stato dell'arte, ma un analisi più estesa. Sfruttando le singole repository che compongono i nostri dataset, la pipeline precedentemente identificata è stata sottoposta a valutazioni mirate per determinarne l'applicabilità in contesti Cross-pProject (CPFP) e Within-Project (WPFP). È importante notare che tali valu-

tazioni sono state condotte esclusivamente su repository che soddisfano due criteri specifici: devono contenere almeno 20 test flaky e presentare uno sbilanciamento a favore dei test non flaky.

Questi criteri, anche se numericamente limitati, sono stati selezionati con una finalità precisa. Il primo criterio, ovvero il requisito di almeno 20 test flaky, è stato adottato per garantire una maggiore solidità dei risultati ottenuti, soprattutto nell'ambito di valutazioni Within-Project. La presenza di un numero minimo di test flaky contribuisce a rendere più affidabili le conclusioni tratte dalla valutazione. Il secondo criterio, lo sbilanciamento a favore dei test non flaky, riflette la realtà dei contesti in cui si trovano le suite di test, dove i test flaky sono in minoranza.

In sintesi, queste scelte metodologiche mirate sono state effettuate per garantire una valutazione accurata e rappresentativa della pipeline, tenendo conto delle condizioni reali in cui operano i test flaky e non flaky all'interno delle suite di test.

Abbiamo inizialmente avviato le valutazioni WPFP in un contesto in cui la prediction su una repository target avviene sfruttando solamente le informazioni che si possiedono di quella repository. Tale valutazione è stata eseguita in maniera iterativa utilizzando il 25%, 50% e 75% dei dati della repository target. Tuttavia per evitare che i risultati siano influenzati dal numero d'istanze su cui testare la pipeline, è stato utilizzato lo stesso test set per ogni iterazione che corrispone al 25% della repository.

Successivamente sono state condotte le valutazioni CPFP, in cui sono stati simulati due scenari, uno in cui non si hanno informazioni pregresse sulla repository target, mentre il secondo scenario prevedeva l'utilizzo di alcune informazioni limitate sulla repository target. Nel primo caso è stato effettuato un confronto tra diverse metodologie, tra cui un approccio classico, il filtro di Burak, modelli locali basati su clustering, modelli locali basati sulla repository, TCA e IG_SM_FS_TCA. Mentre per il secondo scenario essendo che si utilizzano dati etichettati del target è stato utilizzato lo stesso approccio del WBFP in modo tale da effettuare un confronto diretto tra i due.

In questo modo, siamo stati in grado di analizzare il comportamento delle diverse metodologie in contesti con diverse quantità di informazioni sulla suite di test delle repository target, permettendo un'approfondita comprensione delle pipeline identificata.

CAPITOLO 5

Risultati

In questo capitolo, vengono analizzati i risultato ottenuti dai vari esperimenti condotti per ogni domanda di ricerca

5.1 Quanto è efficace un approccio basato sul Machine Learning per il rilevamento della flakiness, utilizzando la classica valutazione train-validation-test?

Il precedente studio condotto da Pontillo et al. [1] ha già dimostrato l'efficacia dell'impiego di matrici statiche per la previsione della flakiness, ottenendo un notevole F1-Score del 70% sul dataset FlakeFlagger. Questo risultato evidenzia chiaramente la superiorità delle metriche da loro sviluppate rispetto a quelle utilizzate da Alshammari et al. [17] nello stesso contesto.

Nel presente lavoro, come evidenziato nella tabella di riferimento 5.1, siamo riusciti a superare i risultati precedenti, raggiungendo un F1-Score del 80%. Questo notevole miglioramento è stato reso possibile grazie all'introduzione di una rigorosa fase di data cleaning e all'analisi di diverse pipeline al fine di identificare quella più efficace. Nel contesto del dataset FlakeFlagger, è stato addestrato il classificatore Ran-

domForest utilizzando il dataset normalizzato tramite la tecnica di normalizzazione Min-Max.

Tabella 5.1: FlakeFlagger Cross Validation

Cross Validation 10 Fold									
Repository	TNF	TF	TP	FN	FP	TN	Pre	Rec	F1
alluxio	64	127	121	6	0	64	100%	95%	98%
java-websocket	67	21	20	1	1	66	95%	95%	95%
ambari	244	52	47	5	1	243	98%	90%	94%
http-request	133	18	15	3	1	132	94%	83%	88%
hbase	246	114	91	23	3	243	97%	80%	88%
elastic-job-lite	505	3	2	1	0	505	100%	67%	80%
spring-boot	7	3	2	1	0	7	100%	67%	80%
okhttp	535	99	65	34	11	524	86%	66%	74%
httpcore	504	15	10	5	2	502	83%	67%	74%
incubator-dubbo	1356	16	9	7	2	1354	82%	56%	67%
hector	89	33	17	16	4	85	81%	52%	63%
logback	639	13	4	9	0	639	100%	31%	47%
orbit	22	4	1	3	0	22	100%	25%	40%
activiti	858	16	0	16	2	856	0%	0%	0%
Achilles	1262	4	0	4	0	1262	0%	0%	0%
assertj-core	1000	1	0	1	0	1000	0%	0%	0%
ninja	282	1	0	1	0	282	0%	0%	0%
undertow	43	5	0	5	1	42	0%	0%	0%
wro4j	1069	14	0	14	1	1068	0%	0%	0%
Total	7	3	404	155	29	8896	93%	72%	81%
AUC (Media)	86%								

Per quanto riguarda invece il dataset IDoFT è stato addestrato sempre il classifi-

catore RandomForest utilizzando il dataset IDoFT prima standardizzato tramite la tecnica della Standardizzazione Z-Score e poi bilanciato tramite l'utilizzo di SMOTE. Il tal caso le prestazioni della pipeline sul dataset raggiungono l'88% di Precision, 61% di Recall e 72% F1-Score. Tale calo di performance non ci sorprende dato il forte sbilanciamento del dataset rispetto a FlakeFlagger. Per motivi di leggibilità e sintesi, la tabella relativa alla cross-validation del dataset IDoFT non è stata inclusa in questo documento. Tuttavia, è possibile consultare la tabella completa nel repository GitHub del progetto, disponibile al seguente indirizzo: Flakiness Detection: An Extensive Analysis

5.2 Quanto è efficace un approccio basato sul Machine Learning per il rilevamento della flakiness in una validazione within-project?

Dopo aver condotto un'attenta valutazione delle metriche proposte da Pontillo et al. [1] e aver identificato la migliore pipeline per la predizione della flakiness, abbiamo proceduto con la sua validazione in un contesto Within-Project (WPFP). In altre parole, abbiamo addestrato la pipeline esclusivamente sui dati della repository in questione al fine di prevedere la flakiness dei futuri test. Gli esperimenti condotti non solo ci hanno consentito di valutare l'efficacia della pipeline in un contesto WPFP, ma ci hanno anche aiutato a determinare la quantità di informazioni necessarie per addestrare con successo la pipeline.

I risultati relativi agli esperimenti condotti sul dataset FlakeFlagger sono riportati nella Tabella 5.2. L'analisi di tali risultati rivela che per le repository incluse nel dataset FlakeFlagger, la validazione WPFP produce risultati eccellenti anche utilizzando solamente il 25% dei dati della repository per l'addestramento. In effetti, se consideriamo la media calcolata su tutte le repository dell'F1-Score e l'Area Under the Curve (AUC), la prima raggiunge l'83% e la seconda raggiunge l'88%. Queste performance aumentano ulteriormente all'87% per l'F1-Score e al 91% per l'AUC quando si utilizza il 75% dei dati della repository.

Tabella 5.2: Within-Project FlakeFlagger

Repository	TNF	TF	WP25_F1	WP25_AUC	WP50_F1	WP50_AUC	WP75_F1	WP75_AUC
java-websocket	67	21	97%	99%	97%	99%	97%	99%
okhttp	535	99	70%	79%	73%	81%	73%	81%
hector	89	33	74%	82%	80%	86%	80%	86%
hbase	246	114	79%	84%	87%	90%	91%	93%
ambari	244	52	95%	95%	94%	95%	94%	95%
Media			83%	88%	86%	90%	87%	91%
Standard Deviation			12%	9%	10%	7%	10%	7%
Min			70%	79%	73%	81%	73%	81%
Max			97%	99%	97%	99%	97%	99%

Tuttavia, è importante notare che tale situazione, come evidenziato nella Tabella 5.3, non si riflette nella stessa misura per le repository incluse nel dataset IDoFT. Qui, per ottenere un F1-Score e un AUC accettabili, è necessario addestrare la pipeline su almeno il 75% dei dati.

La differenza nei risultati tra le repository dei due dataset non sorprende, poiché, se esaminiamo il rapporto tra i test flaky e non flaky all'interno delle repository, possiamo notare che le repository che compongono il dataset IDoFT sono molto più sbilanciate. Questo sbilanciamento contribuisce a spiegare il calo delle performance. Tuttavia, per quelle poche repository che presentano proporzioni simili a quelle del dataset FlakeFlagger, i risultati rimangono simili.

Può essere giusto affermare che la quantità minima di dati su cui addestrare il modello per ottenere ottime prestazioni in un contesto WPFP dipende dall'equilibrio tra test flaky e test non flaky all'interno della repository. In particolare, quando il numero di test flaky rappresenta meno del 10% del totale, è necessario utilizzare almeno il 75% dei dati per addestrare il modello al fine di ottenere prestazioni eccellenti. Tuttavia, se la percentuale di test flaky è superiore al 10%, è possibile raggiungere ottime prestazioni anche addestrando il modello utilizzando solo il 25% dei dati disponibili.

Tabella 5.3: Within-Project IDoFT

Repositry	TNF	TF	WP25_F1	WP25_AUC	WP50_F1	WP50_AUC	WP75_F1	WP75_AUC
hadoop	13485	177	52%	70%	63%	77%	69%	81%
dubbo	1500	139	65%	79%	70%	84%	73%	85%
nifi	11007	138	62%	78%	68%	81%	72%	83%
ignite-3	2055	118	84%	89%	90%	93%	92%	95%
ormlite-core	952	90	88%	93%	89%	92%	91%	93%
admiral	2335	55	35%	63%	37%	65%	40%	67%
adyen-java-api-library	208	45	84%	91%	88%	92%	88%	92%
visualee	91	43	72%	79%	81%	86%	89%	92%
servicecomb-java-chassis	2382	41	87%	94%	91%	96%	92%	96%
DataflowTemplates	655	39	81%	87%	85%	90%	86%	91%
wildfly	873	37	74%	85%	78%	86%	81%	89%
Chronicle-Wire	425	35	27%	60%	41%	67%	45%	69%
typescript-generator	98	34	77%	85%	81%	87%	86%	89%
druid	10699	30	14%	55%	36%	66%	50%	73%
http-request	156	28	89%	93%	92%	95%	94%	97%
Java-WebSocket	391	28	93%	96%	96%	98%	97%	98%
mockserver	2077	27	54%	77%	70%	81%	79%	87%
nacos	1824	26	34%	63%	42%	67%	55%	75%
commons-lang	2686	25	52%	73%	61%	79%	68%	82%
fastjson	1835	25	32%	63%	46%	70%	47%	73%
wro4j	1616	25	15%	57%	36%	66%	36%	66%
Achilles	377	24	86%	94%	93%	96%	93%	96%
aem-core-wcm-components	1068	24	55%	75%	69%	84%	75%	87%
ozone	2456	24	9%	53%	10%	53%	38%	63%
flink	9231	23	29%	62%	39%	65%	47%	69%
Activiti	942	22	66%	83%	72%	87%	80%	89%
vpc-java-sdk	2529	22	43%	77%	55%	82%	61%	85%
graylog2-server	2456	21	42%	67%	56%	74%	75%	86%
jackrabbit-oak	10081	20	56%	74%	70%	84%	73%	88%
Media			57%	76%	66%	81%	71%	84%
Standard Deviation			25%	13%	23%	12%	19%	10%
Min			9%	53%	10%	53%	36%	63%
Max			93%	96%	96%	98%	97%	98%

5.3 Quanto è efficace un approccio basato sul Machine Learning per il rilevamento della flakiness in una validazione cross-project?

Tabella 5.4: CPFP Non Supervisionata

Repository	CPC_F1	CPBF_F1	CPLMC_F1	CPLMR_F1	CPTCA_F1	CPIG_SM_FS_TCA_F1
IDoFT						
Achilles	0%	0%	0%	0%	0%	0%
Activiti	10,3%	0%	0%	0%	6%	2%
admiral	0%	0%	0%	0%	0%	0%
adyen-java-api-library	0%	0%	0%	0%	0%	8%
aem-core-wcm-components	0%	0%	0%	0%	26%	12%
Chronicle-Wire	0%	0%	0%	0%	0%	0%
commons-lang	0%	0%	0%	0%	0%	0%
DataflowTemplates	0%	0%	0%	0%	0%	3%
druid	0%	0%	0%	0%	0%	0%
dubbo	0%	0%	0%	0%	2%	20%
fastjson	0%	0%	0%	0%	11%	5%
flink	0%	0%	0%	0%	0%	0%
graylog2-server	0%	0%	0%	0%	5%	1%
hadoop	0%	0%	0%	0%	0%	0%
http-request	0%	0%	0%	0%	0%	0%
ignite-3	0%	0%	0%	0%	1%	0%
jackrabbit-oak	0%	0%	0%	0%	0%	0%
Java-WebSocket	0%	0%	0%	0%	0%	0%
mockserver	0%	0%	0%	0%	0%	16%
nacos	0%	0%	0%	0%	2%	0%
nifi	0%	0%	0%	0%	0%	0%
ormlite-core	0%	0%	0%	0%	0%	0%
ozone	0%	0%	0%	0%	0%	0%
servicecomb-java-chassis	0%	0%	0%	0%	0%	0%
typescript-generator	0%	0%	0%	0%	0%	0%
visualee	0%	0%	0%	0%	0%	0%
vpc-java-sdk	0%	0%	0%	0%	0%	0%
wildfly	0%	0%	0%	0%	0%	39%
wro4j	0%	0%	0%	0%	0%	0%
FlakeFlagger						
java-websocket	95%	98%	0%	0%	95%	91%

okhttp	0%	17%	0%	0%	19%	7%
hector	0%	0%	0%	0%	0%	24%
hbase	6%	5%	8%	0%	11%	29%
ambari	0%	0%	0%	0%	0%	19%

Un'alternativa all'utilizzo della WPFP (Within-Project Flakiness Prediction) per la previsione della flakiness è la CPFP (Cross-Project Flakiness Prediction), in cui si tenta di prevedere la flakiness di una repository target utilizzando una pipeline addestrata su un insieme di repository sorgenti. Se consideriamo il tipico approccio di machine learning, in cui un modello viene allenato su un set generico di dati per successivamente prevedere nuovi dati, potrebbe sembrare che l'approccio CPFP sia l'opzione ideale per la previsione della flakiness. Tuttavia, come evidenziato nella tabella 5.4, sia nel caso del dataset FlakeFlagger che IDoFT, si riscontrano delle difficoltà nell'applicare la CPFP per prevedere la flakiness.

Queste difficoltà derivano dalla differenza nella distribuzione dei dati tra la repository target e l'insieme delle repository sorgenti utilizzate per addestrare il modello. Questa discrepanza nella distribuzione è ulteriormente accentuata nel nostro contesto, in cui utilizziamo più di venti metriche statiche per la previsione. A titolo di esempio prendendo in considerazione la repository wro4j, nella Tabella 5.5 è riportata la distribuzione dei dati sia per la repository target che per i dati di addestramento, espressi in termini di media e deviazione standard. Se osserviamo l'importanza delle singole caratteristiche per il modello, notiamo che le più rilevanti presentano differenze significative nella distribuzione.

Tabella 5.5: Differenza Distribuzione Sorgente-Target

Features	Mean Sorgente	Mean Target	STD Sorgente	STD Target	Importanza
tloc	11,1	6,7	12,5	5,3	0,104
lcom5	3,2	2,2	9,3	2,5	0,087
eagerTest	1,5	0,8	2,6	1,5	0,084
assertionRoulette	1,5	0,4	3,5	1,0	0,073
mpc	106,2	29,2	385,3	26,8	0,069
cbo	22,9	16,8	44,8	11,2	0,067
rfc	129,6	40,4	415,7	35,1	0,064
wmc	69,7	30,5	130,3	25,9	0,061
loc	535,7	176,0	1081,4	126,9	0,056

lcom2	627,3	70,5	3650,2	128,7	0,055
halsteadVocabulary	1790,7	741,8	4181,7	602,5	0,047
halsteadLength	288,7	218,0	417,3	133,0	0,047
halsteadVolume	2309,2	1471,0	4149,7	1076,0	0,045
assertionDensity	0,1	0,0	0,2	0,1	0,041
conditionalTestLogic	0,9	0,3	2,7	1,3	0,029
tmcCabe	2,1	2,1	1,0	1,0	0,027
mysteryGuest	0,5	0,5	2,4	1,7	0,012
sensitiveEquality	0,1	0,0	0,6	0,1	0,009
spaghettiCode	69,6	0,0	463,3	0,0	0,007
functionalDecomposition	0,1	0,1	0,4	0,4	0,004
classDataShouldBePrivate	0,7	0,0	10,1	0,0	0,004
complexClass	28,6	0,0	132,5	0,0	0,003
resourceOptimism	0,0	0,1	0,2	0,5	0,002
fireAndForget	0,0	0,0	0,2	0,2	0,002
godClass	12,3	0,0	216,4	0,0	0,001
testRunWar	0,0	0,0	0,0	0,0	0,000

Per affrontare questa sfida, abbiamo preso spunto dalla defect prediction, un campo in cui le metriche statiche del codice sorgente vengono ampiamente utilizzate per prevedere difetti. Tuttavia, come nel nostro caso, anche in tale contesto gli approcci Cross-Project si scontrano con il problema della differenza nella distribuzione dei dati. È importante notare che sono state proposte diverse metodologie per mitigare questo problema, tra cui il Filtro di Burak (CPBF), i Modelli locali basati su clustering (CPLMC), i Modelli locali basati su repository (CPLMR) e la TCA (CPTCA). Tuttavia, come evidenziato nella Tabella 5.4, va notato che nessuna di queste tecniche ha portato miglioramenti significativi nelle prestazioni Cross-Project, in quanto presentano diverse problematiche.

Per quanto riguarda il Filtro di Burak, il suo approccio di base consiste nel filtrare i dati di addestramento in base ai dati della repositroy target, utilizzando la distanza euclidea al fine di mantenere solo quelli più rilevanti. Tuttavia, il principale problema di questa tecnica è che la selezione dei dati di addestramento avviene utilizzando i dati della repository target, senza tener conto delle loro etichette. Ciò significa che le istanze flaky nella repository target potrebbero non selezionare necessariamente solo istanze flaky dai dati di addestramento, e le istanze non flaky nella repository

target potrebbero non selezionare solo istanze non flaky dai dati di addestramento. In effetti, per la repository wro4j, come riportato nella Tabella 5.6, i test flaky della repository target non selezionano alcun test flaky dai dati di addestramento. Il che implica l'avere un modello addestrato in maniera errata.

Tabella 5.6: Selezione dei campioni filtro di burak

Test Flaky selezionati da istanze Flaky	0
Test Non Flaky selezionati da istanze Flaky	250
Test Flaky selezionati da istanze Non Flaky	97
Test Non Flaky selezionati da istanze Non Flaky	15063

Il concetto fondamentale alla base dell'uso di modelli locali è la suddivisione dei dati di addestramento in numerosi sottoinsiemi. Nel caso del CPLMC (Cross-Project Local Model with Clusters), questi sottoinsiemi sono rappresentati dai cluster, mentre nel caso del CPLMR (Cross-Project Local Model with Repository Groups), sono rappresentati dalle repository stesse. Questo approccio consente a ciascun modello locale di attribuire maggiore importanza a specifiche caratteristiche rispetto ad altre, in base al gruppo di dati a cui appartengono.

Nel contesto del CPLMC, tuttavia, si presenta un problema significativo legato all'omogeneità dei cluster. Per illustrare questa questione, possiamo prendere in considerazione nuovamente l'esempio della repository wro4j. Come mostrato nella tabella 5.7 è evidente che i cluster creati sono fortemente sbilanciati tra di loro. Ad esempio, il cluster 0 contiene il 75% dei dati di addestramento, mentre il cluster 4 ne contiene solo il 18%, e gli altri cluster non superano nemmeno l'1%. Questo sbilanciamento non solo non contribuisce a mitigare il problema della differenza di distribuzione dei dati, ma è stato riscontrato che per le previsioni sono stati utilizzati solamente i modelli relativi al cluster 0 e al cluster 4.

Tabella 5.7: Cluster Repositroy wro4j

Cluster	TNF	TF	% Training	F1-Score
Cluster 0	208790	1904	76,32%	6%
Cluster 1	866	0	0,31%	
Cluster 2	2832	29	1,04%	9%
Cluster 3	10600	117	3,88%	75%
Cluster 4	48521	713	17,84%	82%
Cluster 5	55	0	0,02%	
Cluster 6	356	0	0,13%	0%
Cluster 7	1264	2	0,46%	100%

Mentre per la CPLMR, la problematica sono due ovvere, le prestazioni dei modelli locali addestrati sulle singole repository e la metodologia di selezione del modello locale da utilizzare per la predizione. In questa metodologia, il modello viene scelto utilizzando la distanza euclidea. Data un'istanza di test da prevedere, viene selezionato il centroide più vicino, e successivamente viene utilizzato il modello associato per eseguire la previsione. Il centroide rappresenta la distribuzione media dei test all'interno di una specifica repository. Tuttavia, simile a quanto accade con il Filtro di Burak, questa scelta basata sulla distanza euclidea può portare a situazioni in cui l'istanza da prevedere, sebbene sia flaky, potrebbe essere molto più vicina alle istanze non flaky di una repository X rispetto alle istanze flaky di una repository Y.

Anche per quanto riguarda la CPTCA e la CPIG_SM_FS_TCA (quest'ultima rappresenta un potenziale miglioramento dell'approccio TCA che abbiamo esaminato), non si riesce a risolvere completamente le sfide legate all'analisi Cross-Project. In questa situazione, si osserva una diminuzione della discrepanza nella distribuzione dei dati tra il set di addestramento e il repository target, ma contemporaneamente si assiste a una riduzione della differenza tra i test flaky e quelli non flaky.

Tutti gli esperimenti precedentemente menzionati condividono un aspetto comune, cioè cercano di affrontare l'approccio Cross-Project in modo non supervisionato, senza tener conto delle etichette dei dati. Questa peculiarità fa sì che il problema

non venga completamente risolto e in alcuni casi, possa addirittura essere accentuato ulteriormente.

Al contrario di tale approcci, l'esperimento CPTrAda in cui si utilizza TrAdaBoost, un metodo di transfer-learning supervisionato come riportato nella tabella 5.8 riesce a ottenere risultati simili alla WPFP. Tuttavia bisogna precisare che a differenza della WPFP in tal caso i dati etichettati della repository target non vengono utilizzati per addestrare il modello ma solamente per filtrare i dati d'addestramento.

Tabella 5.8: CPFP Supervisionata

Repository	CP25_F1	WP25_F1	CP50_F1	WP50_F1	CP75_F1	WP75_F1
IDoFT						
Achilles	91%	86%	91%	93%	91%	93%
Activiti	57%	66%	57%	72%	100%	80%
admiral	42%	35%	42%	37%	57%	40%
adyen-java-api-library	78%	84%	84%	88%	84%	88%
aem-core-wcm-components	0%	55%	80%	69%	91%	75%
Chronicle-Wire	0%	27%	0%	41%	0%	45%
commons-lang	50%	52%	67%	61%	67%	68%
DataflowTemplates	67%	81%	63%	85%	84%	86%
druid	40%	14%	20%	36%	18%	50%
dubbo	61%	65%	61%	70%	68%	73%
fastjson	0%	32%	40%	46%	40%	47%
flink	40%	29%	36%	39%	55%	47%
graylog2-server	29%	42%	75%	56%	75%	75%
hadoop	49%	52%	61%	63%	70%	69%
http-request	83%	89%	83%	92%	83%	94%
ignite-3	84%	84%	89%	90%	87%	92%
jackrabbit-oak	80%	56%	73%	70%	73%	73%
Java-WebSocket	93%	93%	93%	96%	93%	97%
mockserver	73%	54%	73%	70%	83%	79%
nacos	17%	34%	62%	42%	67%	55%
nifi	64%	62%	71%	68%	71%	72%
ormlite-core	88%	88%	90%	89%	90%	91%
ozone	0%	9%	25%	10%	25%	38%
servicecomb-java-chassis	82%	87%	82%	91%	95%	92%
typescript-generator	86%	77%	100%	81%	94%	86%
visualee	76%	72%	91%	81%	96%	89%
vpc-java-sdk	44%	43%	50%	55%	67%	61%
wildfly	71%	74%	74%	78%	71%	81%

wro4j	36%	15%	36%	36%	36%	36%
Media	55%	57%	66%	71%	70%	71%
FlakeFlagger						
java-websocket	100%	97%	100%	97%	100%	97%
okhttp	62%	70%	65%	73%	60%	73%
hector	71%	74%	71%	80%	71%	80%
hbase	74%	79%	71%	87%	81%	91%
ambari	96%	95%	96%	94%	96%	94%
Media	81%	83%	81%	86%	82%	87%

CAPITOLO 6

Discussione e limiti

Parlando dei vincoli del nostro studio, vi sono alcune variabili che potrebbero aver influenzato le nostre conclusioni in modo negativo. Questa parte del lavoro esamina tali variabili e descrive le azioni adottate per ridurne l'effetto sulle nostre conclusioni.

6.1 Minaccia alla validità di costrutto

La principale preoccupazione che solleva dubbi sulla validità del nostro costrutto è la possibile inaccuratezza dei dati utilizzati nella nostra ricerca. Abbiamo basato la nostra analisi su fonti di dati accessibili al pubblico, che sono state precedentemente validate in ricerche passate. Sebbene questo ci conferisca una certa fiducia nella qualità dei dati, è essenziale riconoscere che non possiamo escludere completamente la possibilità di errori o inesattezze, soprattutto quando si tratta di identificare i test instabili. Alcuni test, per esempio, potrebbero non aver manifestato appieno la loro instabilità, creando così una sfida significativa. Queste inesattezze sono più evidenti nel dataset IDoFT, poiché la fonte da cui sono stati estratti i test flaky non forniva informazioni sul numero di esecuzioni delle relative suite di test. A differenza del

dataset FlakeFlagger, dove i creatori hanno eseguito le suite di test di ogni repository 10.000 volte.

Inoltre, un altro aspetto rilevante riguarda l'utilizzo di strumenti automatizzati per calcolare le variabili indipendenti. Siamo consapevoli del potenziale "rumore" che tali strumenti potrebbero introdurre, specialmente quando si tratta di individuare "code smells" e "test smells". Tuttavia, è importante notare che abbiamo dovuto accettare queste limitazioni nella nostra ricerca in quanto ci siamo concentrati su un vasto insieme di dati, rendendo un'analisi manuale impraticabile. Per mitigare questa sfida, abbiamo scelto di utilizzare strumenti consolidati con una reputazione affidabile in termini di precisione.

Per quanto riguarda l'associazione tra classi di test e classi di produzione, abbiamo adottato un approccio basato sulle convenzioni di denominazione. Questa scelta è stata motivata dalla necessità di equilibrare la precisione e la scalabilità. È fondamentale notare che questo approccio potrebbe portare a collegamenti errati in situazioni in cui ci sono classi di produzione con nomi identici ma percorsi diversi. Nel nostro studio specifico, non abbiamo rilevato casi in cui ciò costituisse un problema. Tuttavia, in future repliche del nostro studio in contesti diversi, potrebbe essere necessario tenere in considerazione questa potenziale criticità per migliorare ulteriormente l'efficacia dell'approccio basato sui "pattern matching".

Riguardo agli esperimenti condotti, sono presenti due potenziali minacce che riguardano l'implementazione degli esperimenti e il criterio di valutazione delle prestazioni di ciascun esperimento. Per accrescere la nostra fiducia nell'implementazione, abbiamo eseguito gli esperimenti utilizzando tecniche di preprocessing e algoritmi di apprendimento supervisionato offerti da librerie note come scikit-learn, imblearn e adapt. Tuttavia, non possiamo escludere la possibilità di bug negli strumenti o nella loro configurazione.

Infine, per quanto riguarda la valutazione delle prestazioni, abbiamo scelto di utilizzare l'F1-Score come metrica di valutazione. Questa metrica offre un equilibrio tra precisione e recall ed è stata ampiamente adottata in studi relativi alla flakiness e alla previsione dei difetti in progetti cross-project.

6.2 Minaccia alla validità delle conclusioni

Le minacce alla validità della conclusione sono legate alla relazione tra trattamento e risultato. Per quanto concerne gli esperimenti di machine learning nell'ambito di RQ1, abbiamo effettuato un confronto tra vari algoritmi di apprendimento, impiegando diverse strategie di preprocessing, con l'obiettivo di individuare il modello più efficace. Complessivamente, i risultati hanno mostrato che il modello Random Forest ha ottenuto le migliori prestazioni in termini di F1-Score rispetto agli altri modelli. Sul dataset FlakeFagger, tale modello ha correttamente identificato 404 dei 559 test flaky, mentre sul dataset IDoFT ha individuato 1711 dei 2790 test flaky. È importante notare che la base di verità utilizzata potrebbe non essere perfetta, e alcuni dei falsi positivi potrebbero in realtà essere veri positivi, cioè casi in cui il fallimento è avvenuto durante il processo di identificazione. Purtroppo, potrebbe essere impossibile dimostrare con certezza l'identificazione di tutti i test flaky in un progetto. Tutti i dettagli dei risultati sono disponibili nella repository GitHub del progetto denominato "Flakiness".

Un altro aspetto significativo che merita approfondimento riguarda la strategia di convalida utilizzata per giungere alle nostre conclusioni. Per selezionare la migliore pipeline, abbiamo seguito la procedura tradizionale di convalida train-validation-test, ma abbiamo ulteriormente validato i risultati attraverso l'uso della cross-validation al fine di generalizzare le nostre conclusioni. Per quanto riguarda, invece, la pipeline valutata in un contesto WPFP (RQ2) essendo che essa è sottoposta a un processo di addestramento ripetuto su diverse porzioni della repository per evitare che i risultati fossero influenzati dalla quantità di dati utilizzati per il test, quest'ultimo è rimasto invariato. Inoltre per generalizzare ancora di più i risultati l'attività di training è stata ripetuta per 10 volte, variando ad ogni iterazione i dati di test. Infine, per garantire una comparazione accurata delle prestazioni tra uno scenario WPFP e CPFP supervisionato, in entrambi i casi abbiamo impiegato gli stessi dati di test.

6.3 Minaccia alla validità esterna

Le sfide legate alla validità esterna si concentrano sulla capacità di estendere i risultati ottenuti. La nostra ricerca si è concentrata sui dataset FlakeFlagger e IDoFT, che sono circoscritti a progetti open source sviluppati in Java. È importante notare che questi progetti presentano diversità di obiettivi e caratteristiche, il che ci aiuta parzialmente a mitigare questa sfida. Tuttavia, è innegabile che questa limitazione rappresenti un vincolo della nostra indagine. D'altro canto, va sottolineato che la maggior parte dei dataset relativi ai test flaky riguarda prevalentemente progetti Java, giustificando così la nostra scelta iniziale. Tuttavia, crediamo che il nostro approccio possa essere esteso ad altri linguaggi di programmazione orientati agli oggetti, a condizione che vengano selezionati accuratamente gli strumenti per raccogliere le metriche necessarie. Ad esempio, nel caso di Python, sono stati sviluppati diversi strumenti per individuare code smells e test smells.

Un altro aspetto da considerare riguarda l'applicabilità pratica del nostro approccio. La metodologia utilizzata per collegare i test alle classi di produzione limita l'uso della versione attuale del nostro approccio a progetti che seguono specifiche convenzioni di denominazione. Tuttavia, è fondamentale notare che questa scelta non costituisce un ostacolo all'implementazione pratica del nostro approccio. Gli sviluppatori interessati a utilizzare la nostra soluzione possono configurarla in modo che il processo di collegamento segua le norme o le linee guida comuni nello sviluppo del codice. Questo consentirebbe al nostro approccio di essere alimentato con dati ancora più dettagliati e specifici. In altre parole, le decisioni prese nel nostro studio hanno lo scopo di condurre una sperimentazione su larga scala del nostro approccio, ma in una situazione pratica, l'adozione di una soluzione di collegamento più robusta o personalizzata potrebbe potenzialmente portare a un aumento delle dimensioni dei dati di addestramento, migliorando ulteriormente le prestazioni descritte nel nostro lavoro.

CAPITOLO 7

Conclusioni

Nell'ambito dell'ingegneria del software, i test svolgono un ruolo cruciale, rappresentando il principale mezzo attraverso il quale possiamo assicurare la qualità, l'affidabilità e la stabilità del software che sviluppiamo. Senza di essi, il nostro lavoro sarebbe paragonabile a costruire un edificio privo di una solida struttura, vulnerabile al crollo in qualsiasi momento.

I test ci consentono di verificare che ogni parte del codice funzioni secondo le specifiche previste. Ci aiutano a rilevare errori, bug o problemi di prestazioni prima del rilascio del software al pubblico. Questo è fondamentale, poiché errori o malfunzionamenti possono causare danni finanziari, danneggiare la reputazione aziendale e, in alcuni casi, mettere a rischio la sicurezza dei dati.

Inoltre, i test risultano essenziali per la manutenzione del software. Quando apportiamo modifiche al codice o aggiorniamo il software, eseguiamo il test di regressione che ci consente di verificare che le nuove modifiche non abbiano effetti indesiderati su altre parti del sistema. Questo semplifica notevolmente il processo di sviluppo e manutenzione a lungo termine.

Tuttavia, la flakiness rappresenta una minaccia per la pratica del testing, poiché rende i test poco affidabili. Un test flaky è un test non deterministico che produce risultati di successo o fallimento in modo intermittente, senza alcuna modifica agli

input, all'ambiente di esecuzione o al codice sottoposto a test.

L'approccio più affidabile per identificare i test flaky risulta essere quello delle "ReRuns", ovvero eseguire la suite di test N volte e osservare i risultati dei singoli test. Quando un test restituisce risultati diversi in almeno una delle N esecuzioni, allora può essere considerato flaky. Tuttavia, questo approccio è semplice ma costoso, soprattutto per i grandi sistemi software che dispongono di una vasta suite di test. Di conseguenza, spesso le aziende trascurano il problema della flakiness.

Attualmente, la ricerca si è concentrata sulla ricerca di approcci alternativi meno costosi per l'identificazione dei test flaky, come l'utilizzo del machine learning. Sono stati presentati vari studi che utilizzano una combinazione di metriche statiche e dinamiche (ad esempio, la copertura del codice) [17]. Tuttavia, per calcolare la copertura del codice, è necessario eseguire il test. A questo proposito, Pontillo et al. [1] hanno proposto un nuovo set di metriche puramente statiche per prevedere la flakiness. Il loro approccio è migliore rispetto ai precedenti in quanto riduce i costi di identificazione e consente di individuare i test flaky in modo proattivo prima di inserirli nella suite di test.

Tuttavia, va notato che questi studi hanno principalmente dimostrato la fattibilità di predire la flakiness tramite il machine learning, piuttosto che stabilire se sia un'alternativa valida al metodo delle ReRuns. Pertanto, l'obiettivo principale di questo lavoro è stato estendere il lavoro di Pontillo et al. [1] per valutare se il machine learning rappresenti una valida alternativa.

Per raggiungere questo obiettivo, abbiamo replicato il lavoro di Pontillo et al. [1] su due dataset diversi, ossia FlakeFlagger e IDoFT, con il primo dataset in comune. Dopo una fase di pulizia dei dati da noi introdotta, abbiamo identificato la miglior pipeline, che ha raggiunto un F1-Score dell'80% nel contesto di FlakeFlagger, superando il lavoro precedente. Nel dataset IDoFT, abbiamo ottenuto un F1-Score del 72%. Successivamente, per valutare l'utilità reale, abbiamo effettuato una doppia validazione della pipeline in due contesti differenti: WPFP (Within-Project Flakiness Prediction) e CPFP (Cross-Project Flakiness Prediction).

Il WPFP ci ha permesso di valutare l'utilizzo del machine learning in un contesto in cui si conosce la flakiness della propria suite di test e si desidera prevedere la flakiness dei futuri test. In tal caso, abbiamo non solo valutato le prestazioni in un

contesto WPFP ma abbiamo anche identificato la quantità di informazioni necessarie per ottenere prestazioni accettabili del modello. Se il rapporto tra test flaky e test non flaky nella suite di test è superiore al 10%, è sufficiente addestrare il modello con solo il 25% delle informazioni. Se il rapporto è inferiore al 10%, è necessario utilizzare almeno il 75% delle informazioni.

Tuttavia, anche se l'approccio WPFP ha prodotto prestazioni accettabili, presenta il limite di richiedere la conoscenza della flakiness della propria suite di test, che per molti progetti software risulta sconosciuta.

Al contrario, l'approccio CPFP non ha questa limitazione, poiché cerca di prevedere la flakiness di una repository target utilizzando le informazioni da altre repository. Tuttavia, l'uso di sole metriche statiche relative al metodo di test e al codice di produzione rende difficile l'applicazione delle CPFP, poiché non generalizzano correttamente su repository non conosciute, cioè non utilizzate per l'addestramento. In una validazione CPFP, l'F1-Score è risultato essere dello 0%. Questo problema è dovuto alla differenza di distribuzione dei dati tra i dati di addestramento e la repository target, che aumenta con il numero di metriche statiche utilizzate.

Questo problema è già noto nel contesto della defect prediction, dove, in modo simile al nostro lavoro, vengono utilizzate metriche statiche relative al codice di produzione per prevedere i difetti in un codice sorgente. Sono stati proposti vari metodi, sia supervisionati che non, per affrontare il problema del trasferimento tra progetti. Tuttavia, i metodi non supervisionati che affrontano il problema dell'adattamento del dominio senza utilizzare le etichette dei dati nel nostro caso non hanno portato a miglioramenti. Al contrario, il metodo di transfer learning supervisionato "TrAdaBoost" ha prodotto prestazioni simili a quelle della WPFP. Tuttavia, questo metodo richiede ancora informazioni sulla flakiness della suite di test.

In conclusione, riteniamo che gli obiettivi del nostro studio siano stati raggiunti. Come dimostrato dai lavori precedenti, l'intelligenza artificiale può rappresentare una valida alternativa al metodo delle ReRuns, anche se al momento il suo utilizzo è ancora prematuro.

L'approccio CPFP, basato sulle metriche statiche relative al metodo di test e al codice di produzione, si è rivelato impraticabile senza conoscere la flakiness della repository target. Tuttavia, è importante notare che questo problema è legato alle

metriche utilizzate e potrebbe essere evitato utilizzando approcci differenti come il Natural Language Processing (NLP).

Al contrario, se si dispone di informazioni sulla flakiness della propria suite di test, l'approccio WPFP ha prodotto ottimi risultati. In tal caso, l'adozione del machine learning combinato con le buone pratiche per la scrittura dei metodi di test può rappresentare una valida alternativa al metodo delle ReRuns. Tuttavia, bisogna essere consapevoli che, dati i costi elevati e la difficoltà nel ottenere informazioni sulla flakiness della propria suite di test, l'approccio WPFP risulta spesso inapplicabile nella maggior parte dei casi.

Bibliografia

[1] V. Pontillo, F. Palomba, and F. Ferrucci, "Static test flakiness prediction: How far can we go?" *Empirical Software Engineering*, vol. 27, no. 7, p. 187, 2022. (Citato alle pagine 1, 4, 5, 12, 13, 16, 24, 25, 34, 36 e 51)

[2] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008. (Citato a pagina 3)

[3] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 643–653. (Citato alle pagine 3, 4 e 8)

[4] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840. (Citato alle pagine 3, 4 e 8)

[5] M. Fowler, "Eradicatingnon-determinismintests.martinfowlerpersonalblog(2011)," https://martinfowler.com/articles/nonDeterminism.html. (Citato a pagina 3)

[6] J. Micco, "The state of continuous integration testing@ google (2017)." https://research.google/pubs/pub45880/. (Citato a pagina 3)

[7] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 39–48. (Citato a pagina 3)

[8] M. Cordy, R. Rwemalika, M. Papadakis, and M. Harman, "Flakime: Laboratory-controlled test flakiness impact assessment. a case study on mutation testing and program repair," *arXiv preprint arXiv:1912.03197*, 2019. (Citato a pagina 3)

[9] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 1–23. (Citato alle pagine 3 e 9)

[10] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 403–413. (Citato a pagina 4)

[11] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020. (Citato a pagina 4)

[12] A. M. Memon and M. B. Cohen, "Automated testing of gui applications: models, tools, and controlling flakiness," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1479–1480. (Citato a pagina 4)

[13] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444. (Citato alle pagine 4, 8 e 11)

[14] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 433–444. (Citato a pagina 4)

[15] V. Terragni, P. Salza, and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2020, pp. 69–72. (Citato a pagina 4)

[16] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 385–396. (Citato a pagina 4)

[17] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1572–1584. (Citato alle pagine 4, 8, 12, 24, 34 e 51)

[18] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 2019, pp. 312–322. (Citato alle pagine 4 e 11)

[19] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 492–502. (Citato alle pagine 4, 8 e 12)

[20] I. S. Association *et al.*, "829-1998 ieee standard for software test documentation," Technical report, Tech. Rep., 1998. (Citato a pagina 8)

[21] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*. IEEE, 1997, pp. 264–274. (Citato a pagina 8)

[22] B. Zolfaghari, R. M. Parizi, G. Srivastava, and Y. Hailemariam, "Root causing, detecting, and fixing flaky tests: State of the art and future roadmap," *Software: Practice and Experience*, vol. 51, no. 5, pp. 851–867, 2021. (Citato a pagina 8)

[23] V. Pontillo, F. Palomba, and F. Ferrucci, "Toward static test flakiness prediction: a feasibility study," in *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, 2021, pp. 19–24. (Citato a pagina 8)

[24] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 821–830. (Citato a pagina 9)

[25] A. Vehabovic, "The process of changing out expandable elements in a large-scale web application," 2020. (Citato a pagina 10)

[26] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "A conceptual replication of continuous integration pain points in the context of travis ci," in *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 647–658. (Citato a pagina 10)

[27] Z. Fan, "A systematic evaluation of problematic tests generated by evosuite," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 165–167. (Citato a pagina 10)

[28] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, pp. 540–578, 2009. (Citato a pagina 18)

[29] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Transactions on software engineering*, vol. 39, no. 6, pp. 822–834, 2012. (Citato a pagina 18)

[30] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 343–351. (Citato a pagina 18)

[31] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *IEEE transactions on neural networks*, vol. 22, no. 2, pp. 199–210, 2010. (Citato a pagina 19)

[32] K.-R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf, "An introduction to kernel-based learning algorithms," *IEEE transactions on neural networks*, vol. 12, no. 2, pp. 181–201, 2001. (Citato a pagina 19)

[33] A. Gretton, K. Borgwardt, M. Rasch, B. Schölkopf, and A. Smola, "A kernel method for the two-sample-problem," *Advances in neural information processing systems*, vol. 19, 2006. (Citato a pagina 19)

[34] Y. Khatri and S. K. Singh, "An effective feature selection based cross-project defect prediction model for software quality improvement," *International Journal of System Assurance Engineering and Management*, pp. 1–19, 2023. (Citato a pagina 20)

[35] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997. (Citato alle pagine 21 e 22)

[36] W. Dai, Q. Yang, G.-R. Xue, and Y. Yu, "Boosting for transfer learning," in *Proceedings of the 24th international conference on Machine learning*, 2007, pp. 193–200. (Citato a pagina 21)

Ringraziamenti

Ringraziamenti qui...