UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Mining di Informazioni Strutturate da Bug Report: Una Tecnica Basata su NLP

RELATORE

Prof. **Fabio Palomba**

SECONDO RELATORE

Dott. **Emanuele Iannone**

Università degli Studi di Salerno

CANDIDATO

Stefano Zarro

Matricola: 0152107273

Anno Accademico 2022-2023

I computer sono incredibilmente veloci, accurati e stupidi. Gli uomini sono incredibilmente lenti, inaccurati e intelligenti. L'insieme dei due costituisce una forza incalcolabile - Albert Einstein

Sommario

I bug report sono documenti cruciali per la gestione efficace e tempestiva dei difetti di un software. Tuttavia, la loro comprensione accurata e l'estrazione delle informazioni pertinenti possono risultare complesse a causa della presenza di testo non strutturato e informazioni eterogenee. .

La presente tesi si concentra sull'implementazione di una nuova tecnica, denominata BURBLE, per l'estrazione di informazioni strutturate dai report di bug, con particolare attenzione all'estrazione di codice. Lo scopo della ricerca è quello di contribuire allo sviluppo di soluzioni automatizzate per la gestione delle segnalazioni, al fine di ottimizzare il processo di bug-fixing e garantire un elevato livello di efficienza.

Le attuali soluzioni per la gestione dei report di bug presentano diversi limiti. La comprensione del contesto e l'estrazione di informazioni rilevanti dai testi dei report risultano spesso limitate, in particolare per quanto riguarda il codice correlato al bug segnalato. Per superare tali limiti, l'algoritmo BURBLE proposto in questa tesi adotta un approccio che combina tecniche di analisi del linguaggio naturale (NLP) e tecniche di elaborazione di dati strutturati. L'obiettivo è estrarre informazioni rilevanti dai report di bug in modo più accurato e completo, inclusi i segmenti di codice pertinenti.

Durante la ricerca, sono stati condotti test sul software implementato utilizzando un ampio campione di report di bug. Tuttavia, i risultati ottenuti non sono stati buonissimi. Ciò è probabilmente dovuto a diversi fattori che potrebbero non essere stati pienamente considerati durante l'implementazione. Nonostante i risultati non siano stati soddisfacenti, è importante sottolineare che la tecnica in questione rappresenta comunque un contributo alla ricerca su automatistmi per la gestione delle segnalazioni fornendo una base per futuri miglioramenti e affinamenti della tecnica.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Motivazioni e Obiettivi

Nell'ambito dello sviluppo software, è inevitabile l'insorgere di bug nei programmi scritti, ed è pertanto necessario individuare e correggere tali errori durante i vari processi del ciclo del software. Tuttavia, tale attività risulta spesso costosa e richiede un considerevole effort. In particolare, si stima che le attività di test e di debugging assorbano più di un terzo del costo totale dello sviluppo del software [2]. Essenziali sono per la loro risoluzione i *bug report*, segnalazioni di errori o comportamenti inaspettati che contengono preziose informazioni per la risoluzione finale dei bug presenti nel software, come stacktrace, descrizioni e possibili soluzioni. L'analisi manuale dei bug report può essere lunga e costosa, soprattutto quando si lavora su grandi quantità di dati, per questo motivo, la ricerca si è concentrata sullo sviluppo di diverse tecniche per l'estrazione automatica e la strutturazione di informazioni dai bug report. Queste tecniche utilizzano l'elaborazione del linguaggio naturale e l'apprendimento automatico per analizzare i report e identificare le informazioni rilevanti. Nonostante i vantaggi offerti da queste tecniche, ci sono molti limiti da considerare: ad esempio, l'identificazione automatica delle informazioni può essere non sempre accurata, questo dipende dalla corretta formattazione dei bug report e dalle caratteristiche a loro peculiari.

In questo contesto, l'obiettivo di questa tesi è di sviluppare una tecnica, denominata BURBLE (*Bug Report Bulk Extractor*), per l'estrazione automatica di sezioni di codice sorgente dai bug report. L'approccio proposto si basa sull'utilizzo di tecniche di analisi testuale e

Natural Language Processing per identificare le informazioni pertinenti nei bug report e di tecniche di analisi del codice sorgente per individuare le sezioni di codice collegate ai bug. Lo scopo è di fornire una soluzione che possa ridurre i tempi e i costi di analisi dei bug report e aiutare gli sviluppatori a identificare più rapidamente le sezioni del codice sorgente che necessitano di intervento, nonché fornire uno strumento utile ai ricercatori abilitando a nuovi studi o assistere a quelli già presenti. Infine, la tesi si propone di valutare l'efficacia della tecnica proposta confrontandola con altre tecniche di estrazione di codice dai bug report disponibili in letteratura e di analizzare le eventuali limitazioni e sfide legate all'implementazione di tale tecnica.

1.2 Risultati

La tecnica utilizzata, per la rilevazione di codice attraverso il tool proposto ha riportato un'accuracy del 81% circa. Un'analisi in merito ai risultati ha portato alla conclusione che un possibile limite all'attuale versione di BURBLE può esser causato dalla presenza all'interno dei bug report selezionati di molte istanze di testo che il tool valuta come codice; infatti il dataset preso in esame [1] è composto da molti casi di testo contenente stacktrace relativi ad errori riportanti all'interno delle segnalazioni, che, come tali, non adrebbero valutati come codice bensì come testo.

1.3 Struttura della tesi

Il capitolo 2 affronta il background e lo stato dell'arte, analizzando il contesto dei bug report e presentando una serie di lavori precedenti sul tema dell'estrazione di informazioni da questi. Il capitolo 3 si concentra sulla descrizione della tecnica proposta per l'estrazione di informazioni dai bug report attraverso un sistema di punteggio basato su NLP, illustrando il design e le valutazioni empiriche effettuate. Il capitolo 4 della tesi riporta i risultati raggiunti.

[1] `https://www.kaggle.com/datasets/innocentcharles/bugs-reports`

CAPITOLO 2

Background e Stato dell'Arte

2.1 Bug Report

Per garantire l'efficienza e la qualità del software, numerosi progetti si avvalgono di *bug report* per raccogliere e registrare le segnalazioni di eventuali errori da parte degli sviluppatori, dei tester e degli utenti finali. I bug report, sono documenti utilizzati nel processo di sviluppo software per segnalare problemi o malfunzionamenti riscontrati dagli utenti o dai tester. Tuttavia, con la crescente diffusione del software—soprattutto open source—si sono accumulate ingenti quantità di report di bug. Nel solo caso dell'edizione open source di Eclipse, ad esempio, si registrano in media più di 10,000 segnalazioni ogni anno [3].

Sebbene il grande numero di report di bug possa rappresentare una risorsa preziosa per migliorare la qualità del software, la loro gestione può risultare problematica. La natura dei bug report può cambiare in base all'ente che li gestisce e/o gli utenti che li creano, è possibile riconoscerne la loro struttura, di seguito elencata fra gli elementi che solitamente li accomunano:

- **Id** (1): Un numero o un codice che identifica la singola segnalazione gestita dalla piattaforma di riferimento.
- **Titolo** (2): Testo che riassume i tratti descrittivi del problema riscontrato, a scelta degli utenti.

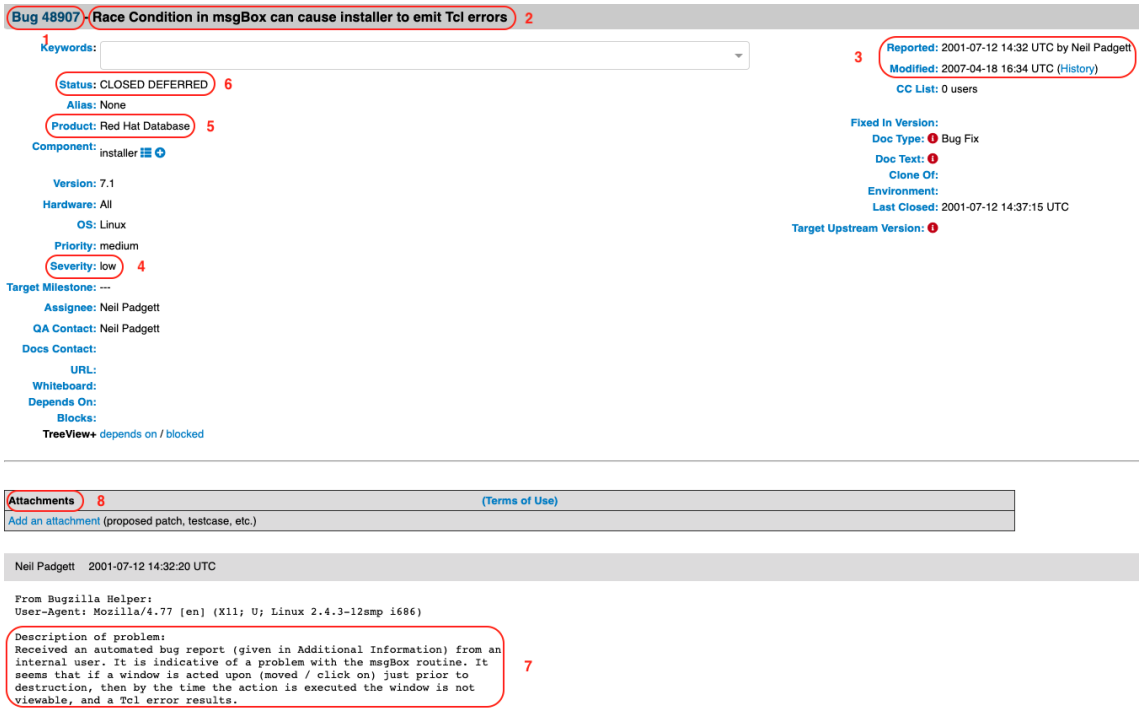

Figura 2.1: Esempio di struttura di un bug report (da `https://bugzilla.redhat.com/`)

- **Data e Ora di segnalazione** (ed eventuale modifica) (3): Riferimento alla data e l'orario dell'avvenuto caricamento della segnalazione all'interno della piattaforma scelta.
- **Gravità** (4): Un sistema di priorità affidato alle segnalazioni, dapprima impostato dal segnalatore, per poi esser eventualemente modificato dal team competente.
- **Team/sviluppatore responsabile** (5): La parte responsabile dell'organizzazione proprietaria del software in questione. Successivamente corretta laddove necessario da chi di competenza.
- **Stato della risoluzione** (6): Indica il punto del processo risolutivo in cui il bug si trova.
- **Descrizione** (7): Un sunto del problema da parte dell'utente, che può elencare a sua volta il codice di riferimento, passi per replicare la problematica riscontrata, il comportamento atteso, il comportamento riscontrato, possibili soluzioni e note aggiuntive.
- **Commenti aggiuntivi**: Commenti post scriptum.
- **Allegati** (8): Eventuali file utili alla comprensione della segnalazione.
- **Bug correlati**: Bug che possono portare, e/o essere portati direttamente o indirettamente a/da altre problematiche.

2.1.1 Il ciclo di vita di un bug report

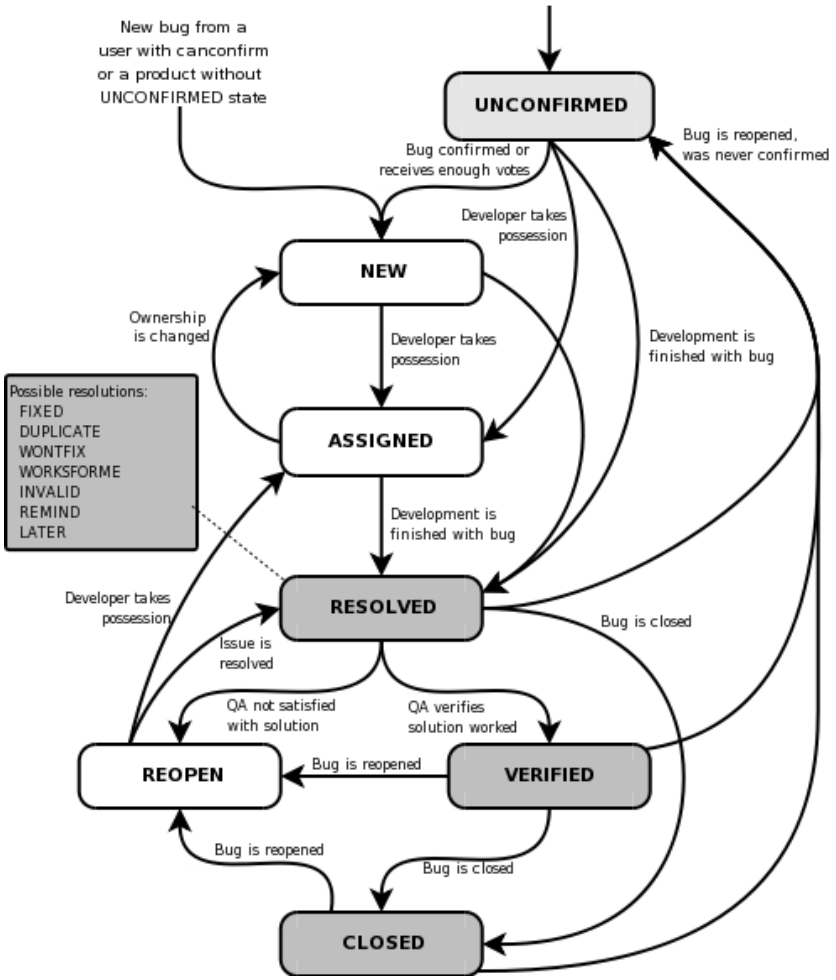

Figura 2.2: Tipico ciclo di vita di un bug report (`https://www.bugzilla.org/docs/2.18/images/bzLifecycle.png`)

Un bug report, come è visibile in Figura 2.2, attraversa vari stati durante il suo ciclo di vita. Tipicamente, non appena il bug viene segnalato, ad esso viene assegnata l'etichetta *UNCONFIRMED* (*"Non Confermato"*), fino a quando un addetto, il *triager*, verifica che si tratta effettivamente di un bug e che non sia duplicato, così da etichettarlo come *NEW* (*"Nuovo"*), e successivamente assegnarlo ad un team/sviluppatore facendo passare il bug allo stato *ASSIGNED* (*"Assegnato"*). In seguito alla risoluzione del bug, esso viene etichettato come *RESOLVED* (*"Risolto"*), e testato da altre figure che approvano la bontà della soluzione; in caso positivo, la segnalazione passerà allo stato *VERIFIED* (*"Verificato"*), altrimenti verrà etichettato come *REOPEN* (*"Riaperto"*) reiterando il ciclo descritto. Lo stato finale è *CLOSED* (*"Chiuso"*), assegnato quando non si verifica nuovamente la presenza di quel bug. Da notare che, quando un bug report passa alla stato *RESOLVED* può attraversare diverse fasi seguenti.

Se uno o più sviluppatori apportano le modifiche di codice necessarie e risolve il bug, lo status verrà modificato da *RESOLVED* a *FIXED* (*"Corretto"*), collegando ad esso i commit (relativi al tool di versionamento del codice utilizzanto dall'organizzazione) che hanno permesso la risoluzione del problema. Se non sussite la volontà di risolvere il bug per qualche motivo (di design, legali, effort, altri vincoli di progetto), lo status verrà impostato su *WONTFIX* (*"Non risolto"*). Se il problema non può essere riprodotto, lo stato verrà cambiato in *WORKSFORME*. Se il triager trova che la segnalazione è un duplicato di una segnalazione di bug esistente, lo stato verrà cambiato in *DUPLICATE* (*"Duplicato"*). Se il bug segnalato dalla segnalazione di bug non è effettivamente un bug, lo stato verrà cambiato in *INVALID* (*"Non valido"*). [4]. Il loro ciclo di vita può subire vari adattamenti in base alle situazioni generate dallo specifico bug e dalle figure coinvolte, difatti, la terminologia adottata può cambiare anche in base alla piattaforma dove la segnalazione viene creata, raccolta e gestita (su Jira ad esempio, la segnalazione può passare allo stato di *"REJECTED"* (*"Rifiutato"*) se lo sviluppatore ritiene che il difetto non sia autentico, oppure allo stato *"DEFERRED"* (*"Differito" o "Posticipato"*) se il bug non ha una priorità abbastanza alta e può essere risolto nella versione successiva. Lo stato differito è anche noto come stato posticipato).

I bug report vengono creati, conservati e utilizzati per lo più nei cosiddetti *Bug Tracking Systems* anche chiamati *ITS* (*Issue Tracking Systems*), progettati per la gestione evolutiva dei problemi, in cui le informazioni vengono raccolte e affinate nel tempo grazie alla collaborazione tra gli sviluppatori e gli stakeholder; i più popolari al giorno d'oggi sono sicuramente *Bugzilla* [1], *Jira* [2] e *Github Issue* [3]. Bugzilla e GitHub sono stati ampiamente studiati dalla comunità di ricerca [5] [6], con un focus sulla qualità delle informazioni nei bug report [7] e sulla predizione di proprietà come la gravità [8], il destinatario [9] e i report duplicati [10] per supportare l'evoluzione e la manutenzione del software.

Jira, invece, è una piattaforma di pianificazione agile che offre funzionalità come Scrum boards, Kanban boards e gestione della roadmap, oltre al issue tracking. Tuttavia, a differenza di Bugzilla e GitHub, Jira ha ricevuto meno attenzione nella ricerca. Questo potrebbe essere dovuto alla mancanza di disponibilità, alla diversità dei dati presenti su Jira o alla sua giovane età [11]. In particolare Bugzilla è un sistema open-source di tracciamento dei problemi utilizzato da numerose organizzazioni, in particolare nel mondo del software libero. Offre una vasta gamma di funzionalità. È stato sviluppato dalla Mozilla Foundation e ora è

[1] `https://www.bugzilla.org/`

[2] `https://www.atlassian.com/it/software/jira`

[3] `https://github.com/`

gestito da Bugzilla. GitHub invece, è una piattaforma di hosting di codice sorgente basata su Git, che consente agli utenti di caricare e condividere il proprio codice e di collaborare su progetti open-source. GitHub offre anche una funzionalità di tracciamento del codice e dei problemi, che permette di raccogliere e organizzare i problemi relativi ai progetti in corso. La sua interfaccia user-friendly e la sua ampia adozione lo rendono un sistema ITS molto popolare tra gli sviluppatori. Jira, d'altro canto, è una piattaforma di gestione progetti e tracciamento problemi sviluppata da Atlassian. È stata progettata per supportare il metodo agile di sviluppo software e offre funzionalità come le board Kanban e Scrum, nonché la possibilità di personalizzare i campi dei problemi. Jira è molto flessibile e può essere adattato alle esigenze specifiche di ogni organizzazione.

2.2 Sull'importanza dei Bug Report

I bug report sono documenti utili a segnalare i problemi riscontrati dagli utenti di un software e contengono informazioni utili per la risoluzione del problema. Inoltre, possono fornire benefici aggiuntivi alla ricerca di automatismi nell'ambito del *triaging* dei bug [12] [13], come l'addestramento di algoritmi per la predizione della priorità, possibili soluzioni e assegnazioni del team competente.

Nello specifico, le applicazioni al fine di velocizzare e/o facilitare la gestione dei bug, sono molteplici; tra le tante, è stata condotta un survey su 327 sviluppatori per giudicare le più importanti su una scala di *Molto importante*, *Importante*, *Neutro*, *Non importante* e *Poco importante* [14] e in base alle loro riflessioni è possibile individuare le seguenti tematiche:

- **Localizzazione dei bug**: sono stati identificati due motivi per cui i professionisti considerano le tecniche di localizzazione dei bug importanti o molto importanti: accelerare la correzione dei bug e aiutare i principianti a familiarizzare con i progetti software. La localizzazione dei bug è utile per accelerare la correzione dei bug in quanto può ridurre lo spazio di ricerca, identificare il proprietario del codice errato, stimare il tempo di correzione dei bug e risparmiare il tempo impiegato nella lettura dei rapporti sui bug. Inoltre, può aiutare i principianti a diventare più familiari con il codice di un progetto software, in particolare in progetti complessi con migliaia di file. Alcuni professionisti, tuttavia, ritengono che la localizzazione dei bug automatizzata non sia sempre affidabile e che non abbiano un grande bisogno di tali tecniche. Questo può essere dovuto alle differenze nelle caratteristiche dei bug e alla loro difficoltà di localizzazione.

- **Assegnazione dei bug**: lo studio evidenzia tre ragioni per cui le tecniche di assegnazione automatizzata dei bug sono considerate importanti dai professionisti:
  - Velocizzare la correzione dei bug. Alcuni degli intervistati hanno sottolineato che le tecniche di assegnazione dei bug possono aiutare a risolvere i problemi in modo più rapido indirizzandoli ai team di sviluppatori adatti. Ciò potrebbe ridurre la comunicazione inutile tra i team e i dipartimenti non di competenza, portando ad una maggiore soddisfazione dei clienti e a minori perdite per l'organizzazione.
  - Risparmiare tempo prezioso per il triaging dei bug.
  - Migliorare la visibilità dei bug non notati. La gestione automatica dei bug può far emergere i problemi che non erano stati notati in precedenza dai team di sviluppo.

    Anche in questo caso, lo studio fa notare alcune scetticità fra i professionisti, non ritenendo le tecniche di assegnazione automatiche dei bug non affidabili o facilmente sostituibili manualmente.

Altre tematiche affrontate sono, la categorizzazione dei bug in modo da poter organizzare il lavoro di risoluzione e identificare le principali aree di intervento. Inoltre, la rilevazione di bug duplicati o simili può aiutare a evitare di ripetere il lavoro di analisi e risoluzione, consentendo di concentrarsi su problemi unici e distinti. È anche utile collegare i bug report ai commit, ovvero alle modifiche apportate al codice, per monitorare lo stato dei problemi e tenere traccia delle soluzioni implementate. Infine, la predizione della risoluzione, della priorità e della gravità dei bug può aiutare a pianificare il lavoro e assegnare le risorse in modo appropriato, in base alla criticità dei problemi. In generale tutte le motivazioni hanno in comune l'ipotesi di poter essere d'aiuto a velocizzare e migliorare l'efficienza del processo seguito per ogni bug, ma al contempo ciò suscita anche scetticismo da parte degli stessi sviluppatori riguardo alla sua efficacia.

2.2.1 Problemi dei bug report

Vari problemi e sfide insorgono nel momento in cui si tenta di estrarre informazioni strutturate dai bug report. In base alla piattaforma utilizzando per il tracking, possono variare le diverse lingue in cui gli sviluppatori comunicano e/o il formato delle segnalazioni, il che rende l'estrazione non standard e di conseguenza più complessa. Altri problemi che possono insorgere dal tipo di linguaggio utilizzato sono:

- **Descrizioni non chiare** In questo esempio, si evidenzia come la descrizione fornita nella segnalazione presenti una certa opacità, dal momento che l'utente lamenta un comportamento inatteso senza tuttavia fornire alcun elemento di dettaglio in merito. Tale mancanza di specificità può rappresentare un ostacolo alla comprensione del problema da parte dei soggetti coinvolti nella risoluzione della problematica, e pertanto potrebbe richiedere un approfondimento della segnalazione stessa al fine di ottenere maggiori informazioni e circostanziare la natura del comportamento anomalo riscontrato.

```
Title: User data not loading
Description: When I navigate to the user profile page, the user
   data is not loading. I'm not sure what's causing this issue.
   Please investigate and fix the problem.
TypeError: Cannot read property 'name' of undefined
   at UserProfileComponent.ngOnInit (user-profile.component.ts:15)
   at callHook (core.js:4690)
   at callHooks (core.js:4654)
   at executeInitAndCheckHooks (core.js:4604)
   at refreshView (core.js:11168)
   at refreshComponent (core.js:12563)
   at refreshChildComponents (core.js:10889)
   at refreshView (core.js:11196)
   at refreshEmbeddedViews (core.js:12517)
   at refreshView (core.js:11170)
```

Listing 2.1: Esempio di descrizione non chiara in un bug report

- **Abbreviazioni, acronimi e common saying** Nell'esempio in questione, si evidenzia come l'utente abbia fatto uso di diversi acronimi, i quali potrebbero generare confusione o difficoltà di comprensione da parte del lettore o del triager qualora non siano noti a coloro che prendono in esame il presente rapporto di errore. Tale circostanza potrebbe rappresentare un ostacolo alla corretta identificazione e alla risoluzione del problema segnalato, in quanto la presenza di termini tecnici non comprensibili potrebbe generare fraintendimenti o errori di interpretazione.

```
Title: API call not working with CORS
Description: When I make a fetch request to the API endpoint, I'm
   getting a CORS error. Specifically, the error message says
   "Access to fetch at 'https://api.example.com/data' from origin
   'https://my-app.example.com' has been blocked by CORS policy: No
   'ACAO' header is present on the requested resource."
fetch('https://api.example.com/data')
   .then(response => response.json())
   .catch(error => {
      console.error('Error:', error);
   });
I've tried adding the ACAO header to the API response, but it's not
   working. This issue is preventing my application from retrieving
   the necessary data from the API.
```

Listing 2.2: Esempio di abbreviazioni e acronimi in un bug report

- **Errori sintattici e grammaticali** In questo esempio, si nota come l'utente abbia commesso diversi errori grammaticali, tra cui l'uso di parole malformate come "*fanction*" e "*gettin*". Questi errori potrebbero rappresentare una difficoltà per le tecniche di riconoscimento del linguaggio utilizzate, impedendo una corretta interpretazione del testo e generando confusione. Si consiglia pertanto di fare attenzione alla corretta scrittura delle parole utilizzate nella segnalazione, al fine di facilitare la comprensione e la risoluzione del problema da parte dei soggetti coinvolti nella gestione del bug.

```
Title: TypeError when using library function
Description: When I call the capitalize fanction from the
   string-utils library, I'm gettin a TypeError. Specifically, the
   error message says "Uncoght TypeError: string_utils_1.capitalize
   is not a function". The capitalize function should take a string
   and retun a capitalzed version of that string, but it's not
   working correctly. This issue is preventing me from properly
   formatting text in my aplication.
```

Listing 2.3: Esempio di errori grammaticali in un bug report

- **Mancanza di correlazioni e/o fonti** Nell'esempio che segue, si evidenzia come l'utente abbia menzionato il codice che genera il comportamento inatteso, ma non abbia incluso i dati che dovrebbero comparire e che potrebbero essere utili per comprendere meglio la natura del problema. Questa mancanza di informazioni potrebbe rappresentare un ostacolo per gli sviluppatori nella ricerca e risoluzione del bug, in particolare se l'errore si verifica solo in presenza di una specifica tipologia di dati. Sarebbe più utile pertanto fornire tutte le informazioni rilevanti riguardanti il contesto in cui si verifica l'errore, al fine di agevolare la comprensione e la risoluzione del problema da parte dei tecnici preposti.

```
Title: Unexpected behavior when filtering data
Description: When I apply a filter to the data table, some of the
   rows are not being displayed correctly. Specifically, some of
   the rows that should be displayed are missing, and some of the
   rows that should be hidden are still being displayed. This issue
   is causing confusion for users of my application.
Here's the code for the filterData function:
{
   const filteredData = data.filter(row => {
      return row.name.toLowerCase().includes(filter.toLowerCase());
   });
   return filteredData; // bug: should be `return
      filteredData.length ? filteredData : data;`
}
```

Listing 2.4: Esempio di mancanza di fonti in un bug report

- **Informazioni incomplete** L'esempio mostrato descrive come l'utente abbia segnalato un errore senza fornire informazioni aggiuntive, come il tipo di immagine, la sezione in cui viene mostrata e il contesto in cui si verifica l'errore. Questa mancanza di dettagli potrebbe rendere difficile per chi analizza la segnalazione comprendere appieno la natura del problema e individuare la causa sottostante.

```
Title: Image not displaying on page
Description: The image on the page is not displaying. This issue is
   preventing users of my application from viewing important visual
   information.
<img src="/images/logo.png" alt="Logo">
The image file is located in the correct directory and has the
   correct file name and file type. Other images on the page are
   displaying correctly.
```

Listing 2.5: Esempio di informazioni incomplete in un bug report

- **Uso di caratteri/tag speciali di markup** (Markdown, HTML, ecc) In questo esempio, si evidenzia come il bug report contenga alcuni caratteri di markup che potrebbero invalidare la corretta visualizzazione della segnalazione o confondere i software di rilevamento automatico degli errori. Questi caratteri possono rappresentare una fonte di confusione per i tecnici preposti alla gestione del bug report e possono impedire una corretta interpretazione del testo.

```
Title: Text formatting not working in Markdown editor
Description: When I use the Markdown editor to format text, some of
   the formatting tags are not working correctly. Specifically, the
   <b></b> and <i></i> tags are not being applied to the text. This
   issue is preventing me from properly formatting text in my
   application.
```

Listing 2.6: Esempio di un bug report contenente caratteri di markup

- **Uso di caratteri di codifiche non comuni** (caratteri giapponesi, russi ecc) Il bug report mostrato in esempio mostra dei caratteri non visualizzati correttamente per via della codifica diversa utilizzata, rendendo così difficile l'intera comprensione del bug report che dovrebbe essere successivamente analizzato con una codifica consona.

```
\u95ee\u9898\uff1a\u975e\u62c9\u4e01\u5b57\u7b26\u7684\u7f16
\u7801\u95ee\u9898\u000d\u000d\u63cf\u8ff0\uff1a\u5f53\u6211
\uff0c\u8fd9\u4e9b\u5b57\u7b26\u5728\u6211\u7684\u5e94
\u7528\u7a0b\u5e8f\u8fd9\u662f\u83b7\u53d6\u8bf7
fetch('https://api.example.com/data')
   .then(response => response.json())
   .then(data => {
      // process data
   })
   .catch(error => {console.error('Error:', error);});
```

Listing 2.7: Esempio di un bug report contente caratteri con codifiche non comuni

Altri tipi di problemi sono dovuti banalmente alla natura delle informazioni non strutturate: difatti questo tipi di elementi sono spesso circondati da commenti, note e altro genere di messaggi scritti in linguaggio naturale che inevitabilmente generano rumore, errori o incomprensioni; tra i più gravi:

- Interruzioni di riga: enumerazioni, liste e stack trace sono spesso separate da caratteri di nuova riga diversi

```
We noticed a problem with the latest update of our software. When
   users try to input text into certain fields, the program doesn't
   handle line breaks correctly. This can cause text to appear
   distorted, making it difficult to read. We tried to reproduce
   this issue on different systems and environments, and it occurs
   consistently.
Here are some details about the issue:
-Users are reporting that line breaks are being inserted in between
   lines of text, making it hard to read.
-We suspect that a recent code change might have caused this issue.
-We found multiple functions in the codebase that handle line
   breaks, so the issue could be originating from any of them.
-To resolve this issue, we suggest the following:
1)Conduct a thorough investigation of the affected code to identify
   any areas that could be causing the problem.
2)Make improvements to the code to handle line breaks correctly.
3)Test the changes to ensure that they don't cause any new issues
   or unexpected behavior.
```

- Confini non limitati: per ogni elementi è difficile individurare il suo inizio e la sua fine dettati spesso da cambi di contesto, spazi bianchi, punteggiatura ecc.

```
   hey dev team
we have a problem with the software it keeps crashing unexpectedly
   we have found an error in the stack trace it looks like this
java.lang.NullPointerException at
   com.example.package.Class.method(Class.java:123) at
   com.example.package.Class2.method2(Class2.java:456) at
   com.example.package.Class3.method3(Class3.java:789) at
   com.example.package.MainClass.main(MainClass.java:101) we have
   tried to reproduce it on different environments and machines but
   it happens all the time
we think it has to do with a null pointer exception that is thrown
   by the Class.method() function causing other errors that
   ultimately leads to a crash of the entire application. here's
   what we suggest to fix it
review the code associated with the Class.method() function to find
   the cause of the null pointer exception
add error handling and exception catching mechanisms to prevent the
   application from crashing
conduct thorough testing to ensure that the error handling and
   exception catching mechanisms are effective and reliable
we understand that fixing this may take some time but we believe
   it's important to the stability of the software
```

2.2.2 Un'indagine sugli attuali software di estrazione di Bug Report

Uno degli esperimenti riusciti sulla realizzazione di un tool capace di estrarre informazioni di tipo strutturato da bug report è INFOZILLA [1]. In questo caso il tool ha estratto informazioni su un dataset di 161,500 bug reports relativi alla Eclipse Software Foundation. Per individuare i vari elementi, il tool è stato sviluppato realizzando diversi filtri, quali:

- **Filtri di patch**

```
Index: PrecisionRectangle.java
===================================================================
RCS file:
   /home/tools/org.eclipse.draw2d/src/org/.../PrecisionRectangle.java
retrieving revision 1.10
diff -u -r1.10 PrecisionRectangle.java
--- PrecisionRectangle.java 21 Jun 2004 19:57:55 -0000 1.10
+++ PrecisionRectangle.java 23 Jun 2004 20:27:25 -0000
@@ -182,6 +182,31 @@
 return this;
+/**
+ * Unions the given PrecisionRectangle with this rectangle and
   returns ...
```

- **Filtri relativi a stack trace**

```
java.lang.Exception
   at java.lang.Throwable.<init>(Throwable.java)
   at org.eclipse.ui.actions.DeleteResourceAction.delete
(DeleteResourceAction.java:325)
   at org.eclipse.ui.actions.DeleteResourceAction.access$0
(DeleteResourceAction.java:305)
   at org.eclipse.ui.actions.DeleteResourceAction$2.execute
(DeleteResourceAction.java:429)
   at org.eclipse.ui.actions.WorkspaceModifyOperation$1.run
(WorkspaceModifyOperation.java:91)
   at org.eclipse.core.internal.resources.Workspace.run
(Workspace.java:1673)
   at org.eclipse.ui.actions.WorkspaceModifyOperation.run ...
```

- **Filtri sul codice sorgente**

```
When using tabs to format, they should be used only for leading
   indents and not to line up columns of parameters. For example:
public class SomeClass {
   public void someMethod() {
      System.out.println("This is a test"
      + "of the formatter");
   }
}
In this code the second line of the println statement would be
   indented using two tabs and then 19 spaces. This would make sure
   that the code lines up no matter what users set their tabs to.
   This is a REALLY important thing for ...
```

- **Filtri su enumerazioni, liste ecc**

```
I am testing the Performance Monitor in RCP. Everything works
   except for one little thing. It may well be a bug. Here are the
   steps to recreate:
SETTING UP AND INSTALLING THE TESTCASE:
1. Start with a fresh install, unzipped to d:\eclipse-SDK-3.0RC3
2. Also unzip the RCP Runtime Binary to d:\eclipse-RCP-3.0RC3
3. Start Eclipse SDK from (1), load the four org.eclipse.perfmsr*
   plug-ins
into your workspace (they are in the org.eclipse.sdk.tests-features
   project on
dev.eclipse.org). You must use the code currently in HEAD.
4. Unzip the attached code to your workspace, import it. It is a
   "hello world" RCP application using the Performance Monitor. It
   is called "helloworld.rcp".
```

Le tecniche utilizzate per ogni filtro di *InfoZilla* sono le seguenti:

- **Implementazione di parser basati su pattern**, con lo scopo di cercare riga per riga un possibile inizio e fine dei pattern identificati, attraverso l'uso di espressioni regolari o altre tecniche simili: supponendo di avere un file di configurazione di un'applicazione che contiene diverse sezioni, ognuna delle quali ha un formato specifico. Utilizzando le espressioni regolari, possiamo definire un pattern per ciascuna sezione e utilizzarlo per estrarre le informazioni di interesse. Ad esempio, se la sezione "database" contiene il nome del database, l'host e la porta, possiamo definire un pattern come *"database\s*=\s*(\w+)\s*:\s*(\d+)\s*@\s*(§+)"* e utilizzarlo per estrarre il nome del database, l'host e la porta.

- **Euristiche che gestiscano interruzioni di riga poco definite**, problema che può sorgere nel momento in cui l'utente creatore del bug report copia ed incolla da altre fonti snippet di testo o codice (patch, stack trace, codice sorgente ecc). Ad esempio, avendo a disposizione un file di testo che contiene un elenco di nomi e indirizzi e-mail, ma alcuni nomi sono scritti in maiuscolo e altri in minuscolo, e gli indirizzi e-mail non sono separati da una virgola. Utilizzando una euristica, possiamo individuare i nomi che iniziano con una lettera maiuscola e terminano con una lettera minuscola, e supporre che la riga successiva contenga l'indirizzo e-mail corrispondente. In questo modo, possiamo facilmente creare un elenco di nomi e indirizzi e-mail corretti.

- **Island Parsing**, una tecnica di analisi sintattica che prevede di analizzare le diverse parti di una frase come se fossero isole separate, piuttosto che analizzare l'intera frase in una sola volta. [15]. Un esempio può essere un file di codice sorgente che contiene diverse funzioni, ognuna delle quali ha una sintassi specifica, possiamo utilizzare la

Element	Accuracy	Precision	Recall
Patches	100.00	100.00	100.00
Stack Traces	98.50	97.08	100.00
Source Code	98.50	98.00	98.88
Enumerations	97.00	99.00	95.19

Tabella 2.1: Performance dei Risultati del tool InfoZilla in % (da [1])

tecnica dell'island parsing per analizzare ciascuna funzione come se fosse un'isola separata e utilizzare le informazioni raccolte per analizzare l'intero file. In questo modo, possiamo identificare più facilmente gli errori e concentrarci sulla loro risoluzione.

I risultati ottenuti alla fine sono stati vicini alla perfezione riuscendo a catalogare e strutturare gli elementi mostrati nella Tabella 2.1 con accuracy, precision e recall altissimi. Per effettuare questa valutazione sono stati campionati casualmente 800 bug reports dal dataset utilizzato. Dopodichè sono stati verificati manualmente i risultati ottenuti dal tool in modo da confrontare la presenza o meno degli elementi categorizzati, notando la presenza dei parametri di *Accuracy*, *Precision* e *Recall* restituiti dalla matrice di confusione ottenuta.

2.2.3 Dataset di Bug Report

Per una corretta validazione dei tool utili all'automazione del processo gestionale dei bug report, è necessario effettuare diversi esperimenti su medie/grandi quantità di dati, al fine di ottenere statistiche che possono rispecchiare una situazione verosimile dell'applicazione dello stesso. Per ottemperare a ciò, i più grandi sistemi di *bug tracking system* mettono a disposizione enormi quantità di bug report con relative informazioni a riguardo, che talvolta sono consultabili in formati strutturati (.csv, .xlsx, .json ecc) o reperibili dalla piattaforma di riferimento (Github Issue, Bugzilla, Jira ecc). Tra i dataset più grandi e popolari troviamo quelli relativi alle organizzazioni presenti in Tabella 2.2

2.3 Altri lavori sull'estrazione di informazioni da bug report

Per affrontare le problematiche accennate sull'analisi dei bug report, molti ricercatori hanno proposto una serie di tecniche per affrontare le sfide introdotte dalla grande quantità di segnalazioni bug a cui un organizzazione deve far capo, come la rilevazione automatica di bug duplicati e la raccomandazione degli sviluppatori per la correzione dei bug. Nei

Organizzazione	**Link dataset**
Eclipse	https://bugs.eclipse.org/bugs
Mozilla	https://bugzilla.mozilla.org
LibreOffice	https://bugs.documentfoundation.org
Apache	https://issues.apache.org/jira/projects

Tabella 2.2: Elenco dataset di bug report

paragrafi successivi, si prendono in considerazione due studi riguardanti l'estrazione di tali informazioni, al fine di migliorare l'efficienza della gestione dei bug. I risultati delle seguenti sperimentazioni su grandi repository di bug, dimostrano la fattibilità di tali tecniche e l'importanza di continuare a investire nella ricerca di soluzioni automatizzate per la gestione dei bug software.

In questo lavoro [16], è stato preso in considerazione un nuovo approccio per rilevare i report di bug duplicati, attraverso la costruzione di un modello discriminativo che prende in input 2 bug report e li confronta per verificarne la similarità. Il modello fornisce un punteggio sulla probabilità che A e B siano duplicati. Tale punteggio viene utilizzato per recuperare i report di bug simili da un repository per l'ispezione dell'utente. È stata verificata l'utilità dell'approccio descritto, su tre repository di bug di grandi dimensioni relativi alle applicazioni open source: OpenOffice, Firefox ed Eclipse. L'esperimento ha mostrato che il modello supera le tecniche esistenti nello stato dell'arte attuale con un miglioramento relativo del 17-31%, 22-26% e 35-43% rispettivamente per i dataset di OpenOffice, Firefox ed Eclipse.

In questo paper [17], si propone un modello unificato basato sulla tecnica di apprendimento del ranking per raccomandare automaticamente agli sviluppatori di affrontare determinati bug report. Il modello combina informazioni estratte da segnalazioni di bug storiche e codice sorgente per una raccomandazione più accurata. Vengono analizzate 16 caratteristiche di similarità per catturare la somiglianza tra una segnalazione di bug e un profilo di sviluppatore. La validazione è stata condotta su un set di oltre 11.000 segnalazioni di bug provenienti da diversi progetti open source, quali: Eclipse JDT, Eclipse SWT e ArgoUML. I risultati mostrano che la combinazione dei due tipi di caratteristiche (basate sull'attività e sulla posizione) migliora le prestazioni del modello rispetto all'uso di un solo tipo di caratteristica. Tra le 16 caratteristiche proposte, scopriamo che la caratteristica "se un programmatore ha toccato un file contenente potenzialmente un bug" è la caratteristica più importante in tutti e tre i dataset.

CAPITOLO 3

BURBLE: una Tecnica di Mining di Bug Report

3.1 Tecnica e design

Una delle sfide più onerose a cui si deve far capo quando si vuole creare un'automazione riguardo il processo gestionale dei bug report, è quello di dover separare le varie parti che compongono la segnalazione. Tra queste c'è sicuramente l'individuazione del codice sorgente, con le sue complicazioni, dovute alle peculiarità del linguaggio di programmazione utilizzato e dal codice descritto. Nello specifico, in questa sezione si andrà a mostrare un approccio a tale individuazione, analizzando la metodologia scelta, le challenge incontrate, la loro risoluzione e i risultati ottenuti, con eventuali ipotesi di miglioramento. È stato essenziale studiare sia la composizione di un generico testo contenente codice, sia la composizione di un generico testo non contenente alcun riferimento a quest'ultimo al fine di individuare le caratteristiche che contraddistinguono entrambi i casi. I tratti individuati che caratterizzano sia la presenza di codice che la sua assenza, vengono di seguito elencate:

- **Sintassi specifica**: la sintassi specifica è una caratteristica comune a tutti i testi contenenti codice. Si tratta di una serie di regole che definiscono la struttura e la grammatica del linguaggio di programmazione utilizzato. Utilizzare la sintassi adeguata è fondamentale per garantire che il codice funzioni correttamente.
- **Elementi speciali**: gli elementi speciali sono caratteri che vengono utilizzati all'interno del codice per delimitare parti specifiche del codice o per specificare valori o

stringhe. Questi elementi possono includere parentesi, graffe, virgolette, apici, e molti altri. Gli elementi speciali sono fondamentali per la struttura del codice e per la corretta interpretazione da parte del compilatore o dell'interprete del linguaggio di programmazione.

- **Parole chiave**: le parole chiave sono parole che vengono utilizzate all'interno del codice per definire strutture di controllo del flusso di esecuzione del codice, come "if", "else", "for", "while", oppure strutture adibite alla creazione di classi, funzioni ecc.

- **Convenzioni di notazione**: le convenzioni di notazione sono regole non strettamente legate alla sintassi del linguaggio di programmazione, ma che rappresentano convenzioni comuni all'interno della comunità di sviluppatori. Queste convenzioni includono l'uso di nomi di variabili significativi e autoesplicativi, l'uso di maiuscole e minuscole per differenziare nomi di variabili o funzioni, e altre regole che rendono il codice più leggibile e facile da comprendere.

- **Economia di spazi**: l'economia di spazi è una caratteristica comune a tutti i testi contenenti codice. Tendenzialmente il codice è scritto sfruttando il minor numero possibile di spazi, ciò non avviene invece all'interno di un testo qualsiasi in cui ricorrendo all'uso di molteplici parole è necessario individuarle separatamente.

- **Segni di interpunzione**: i segni di interpunzione vengono utilizzati all'interno del codice per delimitare parti specifiche del codice o per specificare valori o stringhe. Come per gli elementi speciali, i segni di interpunzione sono fondamentali per la struttura del codice e per la corretta interpretazione da parte del compilatore o dell'interprete del linguaggio di programmazione, tuttavia c'è bisogno di distinguere fra i segni di interpuzione previsti nella sintassi dei linguaggi di programmazione e quelli utilizzati invece all'interno di un testo qualsiasi per rispettare regole grammaticali e/o dare un tono al discorso.

- **Sentimento non positivo**: la sentiment analysis (analisi del sentimento) è una tecnica utilizzata per determinare se un testo risulta essere *neutro*, *positivo* o *negativo*. Nei testi contenenti codice, il sentimento tende ad essere neutro, in quanto il codice non contiene informazioni emotive. Tuttavia, in alcuni casi, può tendere al negativo, ad esempio quando si utilizzano commenti sarcastici o ironici, o si presentano lamentele riguardo a malfunzioni.

L'intero processo di analisi, sviluppo e validazione di BURBLE è avvenuto utilizzando come linguaggio di programmazione *Python*, con le sue relativi librerie:

- Pandas [1]: per manipolare i dati utilizzati, pulirli ed effettuare le operazioni necessarie su di essi.
- Torch [2]: utilizzato per il corretto funzionamento dei modelli di intelligenza artificiale pre-addestrati di *Hugging Face*
- Transformers [3]: per semplificare l'uso dei modelli di *Hugging Face* nel campo del Natural Language Processing (NLP).
- Matplotlib [4]: per ottenere informazioni visuali utili all'analisi dei dati utilizzati e lo sviluppo della tecnica.

Preso in considerazione un dataset in cui ogni istanza rappresenta una riga associata ad un bug report, è stato possibile identificare delle tecniche per il corretto riconoscimento del codice basate sulle caratteristiche precedentemente evidenziate. Le tecniche utilizzate sono molteplici, e tutte contribuiranno in maniera sequenziale, ad assegnare un punteggio ad ogni riga che sarà più alto se il testo in input ha più probabilità di essere codice, più basso altrimenti. Di seguito vengono analizzate nello specifico i metodi adottati:

- **Riconoscimento del linguaggio**
  La tecnica consiste nel riconoscere con un certo grado di affidabilità la lingua utilizzata nel testo. In questo caso è stato preso in considerazione la sola lingua inglese. Per ottemperare a tale tecnica è stato usato il modello di deep learning *xlm-roberta-base-language-detection*, dalla piattaforma *Hugging Face* [5]. Il modello è basato sul modello transformer *XLM-RoBERTa*, che viene pre-addestrato su un grande corpus di testo in diverse lingue e poi ritoccato per la rilevazione della lingua. Questa tecnica consente al modello di acquisire conoscenze linguistiche più ampie e generiche che possono essere utilizzate per diversi compiti di elaborazione del linguaggio naturale. In particolare il modello riporta una precisione del 99.77% e un F1-score del 99.77%, addestrato e validato su un dataset [6] composto da 2 colonne, una delle quali conteneva il testo

[1] `https://pandas.pydata.org/`
[2] `https://pytorch.org/`
[3] `https://pypi.org/project/transformers/`
[4] `https://matplotlib.org/`
[5] `https://huggingface.co/papluca/xlm-roberta-base-language-detection`
[6] `https://huggingface.co/datasets/papluca/language-identification#additional-information`

del quale dover identificare la lingua e l'altra la lingua identificata manualmente; il dataset è stato diviso in 70k record per la parte di training e in 10k record per la parte di validazione.

- **Sentiment Analysis**
  La tecnica consiste nell'analizzare il *sentimento* di un testo, ovvero l'emozione generale associata al testo stesso. In altre parole, si tratta di determinare se il testo esprime un'opinione positiva, negativa o neutra nei confronti di un determinato argomento o oggetto restituendo un risultato *positivo*, *negativo* o *neutro* a seconda del testo con un certo grado di affidabilità. Il modello usato è *twitter-roberta-base-sentiment*, anch'esso dalla piattaforma *Hugging Face* [7] [18]. È un modello di elaborazione del linguaggio naturale basato sull'architettura RoBERTa, addestrato su un corpus di circa 60k di tweet [8] etichettati con informazioni sul sentiment, con l'obiettivo di apprendere una rappresentazione del linguaggio naturale in grado di catturare le caratteristiche semantiche e sintattiche delle frasi utilizzate nei tweet, al fine di poter distinguere tra tweet positivi, negativi o neutri. Il modello utilizza un'architettura Transformer, in cui l'input del testo viene rappresentato come una serie di token che vengono elaborati in modo parallelo da diverse attenzioni e trasformazioni non lineari per catturare le informazioni semantiche del testo.

- **Riconoscimento notazioni** La tecnica usufruisce di una funzione *check_case* che prende in input una stringa testuale. La funzione determina se la stringa segue una convenzione di naming nota come "*CamelCase*", "*snake_case*" o *PascalCase* comunemente utilizzate nella programmazione. Nello specifico, il codice definisce due espressioni regolari: "camel_case" e "snake_case" (in quanto la notazione PascalCase è di per sè contenuta in quella CamelCase). In seguito, la funzione verifica se la stringa "text" segue una delle due convenzioni tramite la funzione "*re.search()*" della libreria Python "*re*" che cerca all'interno della stringa "text" una corrispondenza tra la stringa stessa e l'espressione regolare passata come parametro. Se la corrispondenza viene trovata, la funzione restituisce True, altrimenti restituisce False.

- **Riconoscimento parole chiave** In questo caso, la metodologia usa una funzione *check_keywords* che prende in input una stringa. La funzione determina il numero di volte in cui alcune parole chiave, definite in un file testuale, sono presenti nella stringa. Il file testuale è sta-

[7] `https://huggingface.co/cardiffnlp/twitter-roberta-base-sentiment`
[8] `https://huggingface.co/datasets/tweet_eval/viewer/sentiment/train`

to ottenuto estraendo le informazioni necessarie da una pagina della piattaforma *Github* [9], dove sono presenti oltre 700 parole chiave usate nei linguaggi di programmazione più comuni come *C, C++, C#, Java, JavaScript, PHP, Python* ecc. Inizialmente, la funzione divide la stringa "text" in una lista di parole utilizzando il metodo "*split()*" di Python. Questo permette di analizzare ogni parola singolarmente. Successivamente, la funzione utilizza un ciclo per scorrere tutte le parole chiave contenute nella lista di parole chiave. Infine grazie al metodo *count()* di Python, la funzione restituisce il numero di volte in cui una parola chiave è presente nella lista di parole analizzate. La funzione somma il numero di occorrenze di ogni parola chiave nella lista di parole analizzate.

- **Riconoscimento simboli** La tecnica è stata implementata attraverso una funzione "*check_symbols*", che prende in input una stringa. La funzione determina il numero di volte in cui una serie di simboli specifici comunemente utilizzati nelle varie sintassi dei linguaggi di programmazione sono presenti nella stringa "text" (ad esempio le varie parentesi *, [], ()* o caratteri di uguagliaza come ==, *!=*, >, <, >=, <= ecc). La funzione utilizza un ciclo per scorrere tutti i simboli presenti nella lista. Per ogni simbolo, la funzione utilizza il metodo "*count()*" di Python per contare il numero di volte in cui il simbolo è presente nella stringa. Il risultato del conteggio viene poi sommato di volta in volta in modo da restituire il numero totale di simboli presenti.

- **Riconoscimento segni di interpunzione** Analagomente alle due tecniche precedenti, anche questo metodo adotta una funzione che, preso in input un testo, restituisce il numero di segni di interpunzione quali *, . ! ? :* , seguiti da un caratteri *blank*, presenti all'interno della stringa.

- **Calcolo percentuale degli spazi** Quest'ultima metodologia, consiste nel contare tutti gli *spazi* identificati all'interno di un testo, per restituirne il rapporto rispetto al numero totale dei caratteri.

L'algoritmo presente in 1 prenderà in considerazione una riga di un bug report, restituendo il valore True se la riga presa in considerazione contiene del codice e False altrimenti.

[9] `https://github.com/e3b0c442/keywords`

```
def CLASSIFIER(text,len_min_string_code,
perc_len_min_string_code,perc_detect_language,
point_check_case,perc_spaces,point_perc_of_spaces,
point_sentiment,perc_score_to_words):
    score = 0
    if len(text) > 0:
        if
        len(text) < len_min_string_code
        and check_symbols(text) >= perc_len_min_string_code *
        len(text) or detect_language(text)[1] <
        perc_detect_language:
            score += check_keywords(text)
            score += check_symbols(text)
            score -= check_punctuation(text)
            if check_case(text):
                score += point_check_case
            if percentage_of_spaces(text) > perc_spaces:
                score -= point_perc_of_spaces
            if sentiment_analysis(text)[0] == "positive":
                score -= point_sentiment
                return score_to_words(score, text) >
                        perc_score_to_words
    return False
```

Listing 1: Codice algoritmo BURBLE

La funzione in esame prende in input i seguenti parametri:

- *text*: la riga del bug report da analizzare.
- *len_min_string_code*: lunghezza minima che il testo deve avere affinchè possa essere valutato correttamente come codice o meno (se non la soddisferà verrà valutato come *non codice*).
- *perc_len_min_string_code*: percentuale della lunghezza del testo da confrontare col numero dei simboli presenti al suo interno.
- *perc_detect_language*: percentuale minima che la funzione *detect_language* deve restituire affinchè il testo venga considerato *non codice*.
- *point_check_case*: punteggio da assegnare se il testo rispetta una notazione particolare.
- *perc_spaces*: percentuale degli spazi in rapporto con i caratteri usati.
- *point_perc_of_spaces*: punteggio da assegnare se il testo supera *perc_spaces*.

- *point_sentiment*: punteggio da assegnare nel caso in cui testo risulta avere un sentimento *positivo*.

- *perc_score_to_words*: percentuale minima da rispettare del punteggio ottenuto in rapporto al numero di parole, affinchè il testo venga valutato come codice.

3.2 Alcuni esempi pratici

In questa sezione viene mostrato un esempio di esecuzione dell'algorirmo implementato, passando in input un bug report contenente degli esempi di codice al suo interno. Prendendo in input il seguente bug report:

```
The method FillLayout.computeChildSize assumes that all LayoutData it
   recieves has a computSize method that returns a point. For instance:
FillData data = (FillData)control.getLayoutData ();
if (data == null) {
   data = new FillData ();
   control.setLayoutData (data);}
Point size = null;
if (wHint == SWT.DEFAULT && hHint == SWT.DEFAULT) {
   size = data.computeSize (control
   flushCache); } else {
In this code if the Control has a grid layout attached to it it will
   result in a class cast exception. The control.getLayoutData will
   return a GridData object.Then when you try and call data.computeSize
   it will generate theexception because GridData.computeSize is a void
   method.
```

BURBLE restituisce il seguente ouput (avendo effettuato precedentemente una divisione per righe della segnalazione che vengono passate di volta in volta come input all'algoritmo). Per comodità l'output generato è stato formattato in modo da avere "*risultao - riga in esame*"

```
False - The method FillLayout.computeChildSize assumes that all
   LayoutData it recieves has a computSize method that returns a point.
   For instance:
True - FillData data = (FillData)control.getLayoutData ();
True - if (data == null) {
True - data = new FillData ();
True - control.setLayoutData (data);}
True - Point size = null;
True - if (wHint == SWT.DEFAULT && hHint == SWT.DEFAULT) {
True - size = data.computeSize (control
True - flushCache); } else {
False - In this code
False - if the Control has a grid layout attached to it
True - it will result in
False - a class cast exception. The control.getLayoutData will return a
   GridData object. Then when you try and call data.computeSize
False - it will generate the exception because GridData.computeSize is a
   void method.
```

CAPITOLO 4

Valutazioni empiriche preliminari

4.1 Contesto

È stato preferito realizzare un tool che prendendo in input il testo della segnalazione, riuscisse ad individuare le sezioni di tale testo contenente codice e quali no.

Per la costruzione di tale tool è stato preso in considerazione il dataset [1] della *Eclipse Software Foundation* contenente le descrizioni complete relative a più di 85,000 issues. Il dataset presenta 10 colonne, descritte come in Tabella 4.1

Prima di procedere allo sviluppo dell'applicativo è stato analizzato in dettaglio il dataset e i suoi relativi dati in modo da raccogliere informazioni utili per la preparazione del lavoro. Dalle analisi si è evinto che:

- Tutti i bug report sono scritti in lingua inglese.
- Una buona parte dei bug report contiene del codice scritto, o un riferimento ad essa.
- Le descrizioni delle segnalazione rappresentano i caratteri di nuova riga con un "***;***"che talvolta vengono ripetuti.
- Le descrizioni delle segnalazione includono i caratteri ***\t*** all'interno del testo (come tabulatori).

[1] `https://www.kaggle.com/datasets/innocentcharles/bugs-reports`

Tabella 4.1: Descrizione colonne dataset *eclipse_platform*

in cui la colonna *Valid* identifica il numero di istanze valide della colonna corrispondente all'interno del dataset, la colonna *Mismatched* il numero di istanze che presentano una discrepanza o un'irregolarità e la colonna *Missing* indica il numero di istanze mancanti.

Nome	Valid	Mismatched	Missing	Descrizione
Issue_id	85.2k	0	0	Identificativo della issue
Priority	85.2k	0	0	Priorità assegnata al bug suddivise in P1, P2, P3, P4, P5
Component	85.2k	0	0	Il componente che riguarda il problema
Duplicated Issue	14.4k	0	70.9k	Identificativo della issue riscontrata come duplicata (se esistente)
Title	85.2k	0	0	Titolo assegnato al bug report
Description	85.0k	0	129	Descrizione della segnalazione
Status	85.2k	0	0	Stato della issue tra *RESOLVED*, *VERIFIED* e *FIXED*
Resolution	85.2k	0	0	Stato della risoluzione della issue
Version	76.0k	9184	9	Versione del componente in questione
Created_time	85.2k	0	0	Data di creazione della segnalazione

4.2 Preparazione dei dati

L'obiettivo di questa fase è di preparare i dati per la successiva fase di creazione del tool di estrazione. Una maggiore qualità dei dati contribuisce sicuramente a raggiungere risultati migliori. È stato per cui necessario selezionare dapprima le feature da considerare per il lavoro da svolgere, e poi la pulizia dei dati stessi, al fine di ottenere un input standard e omogeneo. In questo caso sono stati presi in considerazione solo i campi contenente l'identificativo di ciascun bug report (il campo *Issue_id*) e la descrizione dello stesso (*description*), ritenendo gli altri poco utilizi al fine dell'estrazione.

Successivamente, è stato utile trovare un'euristica per separare ogni bug report in righe/contesti diversi di modo da andare a verificare la presenza del codice all'interno di ognuna delle sezioni identificate, per ottenere una precisione migliore e una localizzazione più accurata del codice.

Tabella 4.2: Valori presi in condiderazione per l'ottimizzazione

Parametro	**Valori da testare**
len_min_string_code	3, 4, 5, 6, 7, 8
perc_len_min_string_code	0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
perc_detect_language	0.7, 0.8, 0.9
point_check_case	0.5, 1, 1.5, 2, 2.5
point_perc_of_spaces	0.5, 1, 1.5, 2, 2.5
perc_spaces	0.05, 0.1, 0.15, 0.2, 0.25
point_sentiment	-0.5, -1, -1.5, -2, -2.5
perc_score_to_words	0.6, 0.7, 0.8, 0.9

4.2.1 Pulizia dei dati

Come visto nella Tabella 4.1, selezionando solo le colonne *Issue_id* e *description* non si avranno problemi di dati nulli o mancanti. Considerando però le problematiche elencate, è stato doveroso eliminare il rumore generato dalla presenza di *;* e $\backslash t$ non desiderati. A tal proposito, la problematica in questione (riguardante la presenza dei *;* come carattere di nuova riga) è stata sfruttata a vantaggio del sistema. Approfittando difatti del carattere in questione, si è riuscito ad ottenere una separazione più o meno affidabile dei bug report divisi per righe.

4.3 Validazione e ottimizzazione

Le variabili di input della funzione: *bug_report*, *len_min_string_code*, *perc_len_min_string_code*, *perc_detect_language*, *point_check_case*, *point_perc_of_spaces*, *perc_spaces*, *point_sentiment*, *perc_score_to_words*, sono state analizzate attraverso la loro combinazione al fine di ottenere quella che restituisse i risultati migliori sui valori presenti in Tabella 4.2:

la funzione 1 è stata infatti (per ogni combinazione) applicata all'intero dataset in esame e, conservando per ognuna di essa i valori relativi alle combinazioni testate e i loro risultati in termini di *false positive*, *true positive*, *false negative* e *true negative* è stato possibile visionare quella che ha performato meglio. Il numero totale delle combinazioni è pari a 315000, ognuna della quale applicata ad un dataset opportunamente costruito campionando randomicamente 300 bug report dal dataset *eclipse_platform.csv*, divisi in record singoli come effettuato precedentemente riga per riga, ottenendo così un dataset finale per la validazione di 6724 record. Ogni record è stato in seguito etichettato e ispezionato manualmente come *contenente*

Tabella 4.3: Valori parametrici finali

Parametro	**Valore**
len_min_string_code	8
perc_len_min_string_code	0.3
perc_detect_language	0.7
point_check_case	0.5
point_perc_of_spaces	0.5
perc_spaces	0.25
point_sentiment	-0.5
perc_score_to_words	0.9

codice o meno. Dal dataset risultante è emerso che esso risulta sbilanciato, in particolare i casi favorevoli (record contenenti codice) sono 903 (*13.43%*) a fronte dei casi sfavorevoli di 5821 record (*86.57%*).

Al fine dell'analisi, il risultato migliore è pervenuto con i valori in Tabella 4.3 ed i relativi risultati in Tabella 4.4.

Tabella 4.4: Parametri combinazioni e matrice di confusione

Parametro	**Valore**
true_positive	560
true_negative	4908
false_positive	913
false_negative	343
Accuracy	81.32
Precision	38.01
Recall	62.01
F1-score	47.13

CAPITOLO 5

Conclusioni

La tesi si focalizza sull'estrazione automatica di codice dai bug report, al fine di agevolare la ricerca e automatizzare la gestione dei bug, nonché accelerarne la risoluzione. Il problema affrontato riguarda la complessità nell'individuare e recuperare i frammenti di codice rilevanti all'interno dei report, a causa della presenza di informazioni eterogenee e non strutturate. Per affrontare questa sfida, è stato sviluppato un software appositamente dedicato all'estrazione automatica del codice dai bug report. Il software sfrutta approcci basati su tecniche di elaborazione del linguaggio naturale e algoritmi di apprendimento automatico per identificare i frammenti di codice pertinenti all'interno dei report.

Dall'analisi delle metriche presentate nella Tabella 4.4, emergono i risultati ottenuti da BURBLE. L'accuratezza del 81.32% evidenzia la capacità del sistema di individuare correttamente i frammenti di codice rilevanti all'interno dei report. Tuttavia, la metrica di precisione del 38.01% indica che solo una percentuale limitata dei frammenti di codice estratti è stata identificata correttamente.

Questi risultati rappresentano una solida base su cui incentrare futuri sviluppi. Un possibile miglioramento potrebbe riguardare l'ottimizzazione dell'algoritmo di estrazione del codice, al fine di ridurre il numero di falsi positivi e falsi negativi, migliorando così la precisione e il recall complessivi del sistema. Inoltre, sarebbe interessante esplorare ulteriori tecniche di elaborazione del linguaggio naturale e di apprendimento automatico, o modificare quelle attuali al fine di perfezionare la capacità del software nel riconoscere e comprendere i frammenti di codice in maniera più coerente ed esaustiva.

Bibliografia

[1] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 27–30, 2008. (Citato alle pagine v, 13 e 15)

[2] T. Xie, L. Zhang, X. Xiao, Y.-F. Xiong, and D. Hao, "Cooperative software testing and analysis: Advances and challenges," *Journal of Computer Science and Technology*, vol. 29, no. 4, pp. 713–723, 2014. (Citato a pagina 1)

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proceedings of the 28th international conference on Software engineering*, pp. 361–370, 2006. (Citato a pagina 3)

[4] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, vol. 58, no. 2, pp. 1–24, 2015. (Citato a pagina 6)

[5] N. Serrano and I. Ciordia, "Bugzilla, itracker, and other bug trackers," *IEEE software*, vol. 22, no. 2, pp. 11–13, 2005. (Citato a pagina 6)

[6] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillere, J. Klein, and Y. Le Traon, "Got issues? who cares about it? a large scale investigation of issue trackers from github," in *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, pp. 188–197, IEEE, 2013. (Citato a pagina 6)

[7] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 34–43, 2007. (Citato a pagina 6)

[8] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu, and I. Illahi, "Deep neural network-based severity prediction of bug reports," *IEEE Access*, vol. 7, pp. 46846–46857, 2019. (Citato a pagina 6)

[9] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 111–120, 2009. (Citato a pagina 6)

[10] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pp. 52–61, IEEE, 2008. (Citato a pagina 6)

[11] L. Montgomery, C. Lüders, and W. Maalej, "An alternative issue tracking dataset of public jira repositories," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 73–77, 2022. (Citato a pagina 6)

[12] G. Murphy and D. Cubranic, "Automatic bug triage using text categorization," in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pp. 1–6, Citeseer, 2004. (Citato a pagina 7)

[13] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, "Automatic bug triage using semi-supervised text classification," *arXiv preprint arXiv:1704.04769*, 2017. (Citato a pagina 7)

[14] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 836–862, 2020. (Citato a pagina 7)

[15] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 13–22, 2001. (Citato a pagina 14)

[16] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 45–54, 2010. (Citato a pagina 16)

[17] Y. Tian, D. Wijedasa, D. Lo, and C. Le Goues, "Learning to rank for bug report assignee recommendation," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10, IEEE, 2016. (Citato a pagina 16)

[18] F. Barbieri, J. Camacho-Collados, L. Neves, and L. Espinosa-Anke, "Tweeteval: Unified benchmark and comparative evaluation for tweet classification," *arXiv preprint arXiv:2010.12421*, 2020. (Citato a pagina 20)