Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Triennale in Informatica

Tecniche di deep learning per il riconoscimento di errori nei programmi

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Matteo Vannucchi

Sessione I
Anno Accademico 2021-2022

Abstract

Il ruolo dell'informatica, in un mondo in progressiva digitalizzazione di ogni singolo aspetto della vita dell'individuo, è ormai diventato chiave del suo funzionamento. Con l'aumentare della complessità del codice e delle dimensioni dei progetti, il rilevamento di errori diventa sempre di più un'attività difficile e lunga. Meccanismi di analisi del codice sorgente tradizionali sono esistiti fin dalla nascita dell'informatica stessa, e il loro ruolo all'interno della catena produttiva di un team di programmatori non è mai stato cosi fondamentale come lo è tuttora. Questi meccanismi di analisi però non sono esenti da problematiche: il tempo di esecuzione su progetti grandi e la percentuale di falsi positivi possono infatti diventare un grosso problema. Per questi motivi meccanismi fondati su *Machine Learning*, e più in particolare *Deep Learning*, sono stati sviluppati negli ultimi anni. Questo lavoro di tesi si pone quindi l'obbiettivo di esplorare e sviluppare un modello per il riconoscimento di errori in un qualsiasi file sorgente scritto in linguaggio sia C sia C++.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Capitolo 1

Introduzione teorica

1.1 Code2Vec

1.1.1 Meccanismo di attenzione

1.1.2 AstContext

Capitolo 2

Dataset

In questo capitolo tratteremo la generazione del dataset posto alla base del modello che andremo a creare poi nel Capitolo 3. Vederemo prima il dataset originale utilizzato e poi come è stato aumentato tramite l'utilizzo di ulteriori analizzatori statici per migliorarne la precisione delle rilevazioni, andando a ridurre il numero di falsi positivi. Verrà poi presentato come le rilevazioni degli analizzatori statici sono utilizzate per la associazione fra un *code snippet* e il relativo errore, poi come da quest'ultimo venga ricavato il codice in formato di *ast context vector*.

2.1 Dataset originale

Come detto in precedenza questo dataset non è stato generato partendo da zero ma facendo riferimento al dataset creato da [1]. Il dataset consiste di circa 3000 progetti di GitHub, scritti in linguaggi C e C++, che rispettano due requisiti: hanno una licenza ridistribuibile e hanno almeno 10 stelle. Il secondo requisito ci serve per garantire che i progetti all'interno del dataset soddisfino dei requisiti di qualità, infatti come precedenti studi hanno mostrato (come ad esempio [2]) si può utilizzare il numero di stelle su GitHub come un *proxy* per la qualità del codice stesso.

Il dataset contiene per ogni progetto una serie di analisi effettuate: l'analisi di Doxygen che estrae le coppie codice-commento e l'analisi di Infer che produce un report di analisi statica degli errori. Visto l'utilizzo che ne sarebbe stato fatto di questo dataset l'analisi di Doxygen è stata scartata. In Figura 2.1 si può vedere la struttura tipica di uno dei circa 3000 progetti presenti.

Come si può notare ogni progetto contiene anche un Makefile, elemento fondamentale perché gli analizzatori statici che andremo ad aggiungere spesso richiedono l'esistenza di

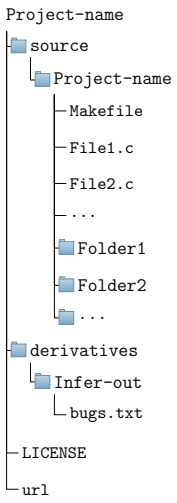

Figura 2.1: La struttura della directory di un progetto del dataset iniziale

un Makefile funzionante.

2.2 Analizzatori di codice statici

Un'analizzatore di codice è un programma che prende in input uno o più file e genera un report degli errori, cioè una lista di coppie del tipo <Errore, Posizione>. Di questi analizzatori ne esistono due macro categorie: statici e dinamici. Gli analizzatori statici sono programmi che effettuano controlli solo sul codice a livello testuale e che quindi non eseguono in nessuna maniera il codice. Gli analizzatori dinamici sono invece analizzatori più complessi che effettuano controlli a *run-time* andando quindi ad'eseguire il codice stesso.

Gli analizzatori non sono però perfetti, infatti nell'insieme degli errori trovati si possono spesso trovare dei falsi positivi, cioè frammenti di codice segnalati come erronei ma che in realtà non presentato nessun tipo di problema. Scopo appunto del dataset aumentato è quello di ridurre il numero di falsi positivi.

2.2.1 Analisi a livello di progetto

La maggior parte degli analizzatori statici inoltre è in grado di lavorare a livello di progetti, andando quindi a risolvere correttamente gli *include* (nel caso di C e C++), e quindi generando un output più significativo. Alcuni di questi per far ciò hanno bisogno di un

file che viene chiamato *compilation database* che mantiene informazioni sulla compilazione dei file del progetto. Per soddisfare questo requisito esistono strumenti appositi che utilizzano il Makefile per generarlo, nel caso di questo lavoro è stato utilizzato un programma chiamato Bear.

2.2.2 Analizzatori ulteriori utilizzati

Come analizzatori statici ulteriori da aggiungere in più a Infer, di cui ogni elemento del dataset ha già l'analisi sua associata, sono stati scelti i seguenti tre:

- L'analizzatore Cppcheck che, a detta degli autori, ha come scopo principale il ridurre il numero di falsi positivi.
- Il compilatore GCC che, nonostante sia un compilatore vero e proprio, ha anche funzionalità per l'analisi statica dei programmi attraverso la flag *-fanalyzer*.
- Il compilatore Clang che attraverso un suo tool chiamato Clang-Check è in grado di effettuare analisi statiche.

Non sono invece stati usati analizzatori dinamici, questo perché il loro utilizzo in modo automatizzato è un operazione complicata se non quasi impossibile. Infatti quasi tutti i programmi prendono o dei parametri all'esecuzione o degli input durante l'esecuzione, ma fornire questi dati in modo consistente e sensato per il programma e in modo automatizzato rende il tutto veramente difficile.

L'utilizzo di essi però potrebbe portare a risultati molto interessanti poiché parte dei falsi positivi degli analizzatori statici deriva dal non poter decidere se frammenti di programmi sono o non sono eseguiti e quindi gli analizzano tutti. Può infatti succedere che se in un frammento di programma che non viene sicuramente mai eseguito c'è un errore, l'analizzatore statico lo riferisce mentre quello dinamico, correttamente, no.

2.3 Utilizzo efficace di processori multicore

L'ultimo argomento da discutere prima di illustrare i passaggi della generazione del dataset è il tempo di esecuzione. Vista la mole di progetti e le loro dimensioni non irrilevanti se eseguissimo in modo *naive* la generazione del dataset avremmo tempi di analisi che possono estendersi anche a periodi di giorni. Dal momento che il processore utilizzato per la generazione del dataset è un processore multicore è stato deciso di ridurre i tempi di esecuzione delle fasi della generazione sfruttando ciò. Python attraverso la sua libreria *multiprocessing* permette infatti di eseguire la computazione in processi diversi,

andando a ridurre drasticamente il tempo delle operazioni. Quindi tutte le operazioni di seguito descritte, anche non facendone più menzione, saranno eseguite in questa maniera.

2.4 Prima fase: generazione dei report degli errori

La prima fase della generazione del dataset consiste quindi nell'utilizzare i tre analizzatori scelti per generare ulteriori report degli errori, in particolare:

- Per eseguire l'analizzatore di GCC vengono prima raccolti tutti i file sorgenti del progetto, cioè tutti quei file che terminano con ".c", ".cpp" o ".h". Una volta fatto ciò viene eseguito il seguente comando:

  $ *gcc -fanalyzer -Wall <files> 2>gcc-bugs.txt*

  Il prodotto di questo comando sarà un unico file contente tutti gli errori e la loro posizione indicata con il percorso relativo del file e il numero sia della riga sia della colonna.

- Clang-check invece può essere eseguito su una cartella e quindi si occupa lui di trovare i file da analizzare. Non viene però utilizzato in questa modalità perché nel file di output finale al posto del percorso relativo dei file viene utilizzato solo il nome di questo, ma può succedere che in progetti grandi si abbiano file chiamati uguali ma in cartelle diverse. Per risolvere questo problema viene eseguito individualmente su ogni file tramite il comando:

  $ *clang-check --analyze -p compile_commands.json <file>*

  Come si può notare utilizza un file chiamato compile commands.json che è il file che è richiesto da certi analizzatori statici, come già detto nella sottosezione 2.2.1. Gli output generati dall'esecuzione di questi comandi vengono poi processati andando a sostituire i nomi dei file con il loro percorsi relativi, e poi uniti tutti insieme.

- Per finire viene poi eseguito cppcheck che invece non ha bisogno di nessun aggiustamento e si può eseguire direttamente su tutta la cartella contenente i sorgenti con il seguente comando:

  $ *cppcheck <cartella_sorgenti> --output-file=cppcheck-bugs.txt*

In Figura 2.2 possiamo vedere quanti errori sono stati generati da ogni singolo analizzatore.

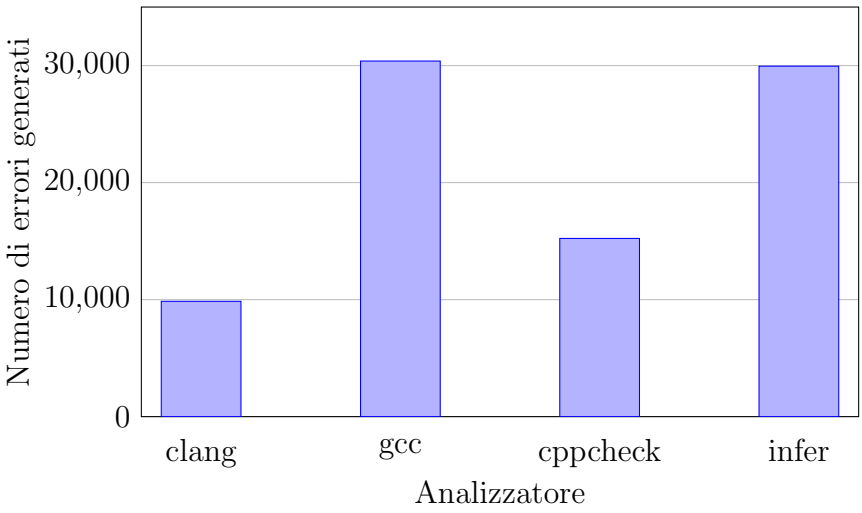

Figura 2.2: Numero di errori generati da ogni analizzatore

2.5 Seconda fase: aggregazione dei report degli errori

Dopo la prima fase descritta nella sezione precedente avremo come risultato quattro report di errori in file separati. Questi report si distinguono per due caratteristiche principali: la struttura del file e la nomenclatura degli errori. Per poter andare ad'utilizzare questi risultati, e fare quindi l'aggregazione di essi, dovremo effettuare due trasformazioni: un *parsing* e una *normalizzazione*.

2.5.1 Parsing dei report

Il *parsing* è l'analisi di un dato in forma testuale per identificarne le sue componenti principali dove, in questo caso, sono la tipologia di errore e la sua posizione. Nel nostro caso è possibile eseguire il parsing tramite delle specifiche *regex* che, avendo diversi formati di file, saranno diverse per ognuno degli analizzatori. Il risultato del parsing sono quindi tanti record nella forma $\langle errore,\ posizione \rangle$, dove la posizione indica sia il percorso del file ma sia anche la riga e la colonna dell'errore.

2.5.2 Normalizzazione

Per *normalizzazione* si intende il processo di uniformare ad'un unico spazio di valori i dati forniti. Questo fase è fondamentale poiché i vari analizzatori forniscono lo stesso tipo di errore sotto nomi diversi. Per fare un esempio possiamo guardare la Tabella 2.1 che riassume le diverse nomenclature per il tipo di errore 'memory leak'.

Forma normalizzata	Infer	Clang	Cppcheck	GCC
Memory leak	MEMORY_LEAK	unix.Malloc, ...	memleak, memleakOnRealloc, ...	Wanalyzer-malloc-leak

Tabella 2.1: Tabella delle diverse nomenclature per l'errore 'memory leak'

Notiamo inoltre un concetto fondamentale: analizzatori diversi producono analisi a granularità diverse. Si può osservare granularità maggiore, per il tipo di errore 'Memory leak', da parte di Cppcheck e Clang nella Tabella 2.1. Infatti tutti e due definiscono più tipologie di errori che però, per convezione di questo progetto, vengono raggruppate in un'unica macro categoria. Al contrario ci sono invece casi in cui un analizzatore non ha sensibilità sufficiente per distinguere fra due o più categorie di errori, in questa situazione un errore di quel tipo viene normalizzato in un errore per ogni categoria che potrebbe rappresentare, si può vedere ciò nella Tabella 2.2. Nella eventualità quindi che Clang riferisca un errore di tipo 'unix.Malloc' dopo la fase di normalizzazione avremo due errori nella stessa posizione: uno di tipo 'Memory leak' e uno di tipo 'Use after free'.
Per effettuare la normalizzazione è stata quindi sviluppata una tabella che associa ad'ogni forma normalizzata degli errori le forme definite dagli analizzatori usati. Questa tabella è stata poi utilizzata come dizionario per convertire le tipologie di errori.

Forma normalizzata	Clang
Memory leak	unix.Malloc, ...
Use after free	unix.Malloc, ...

Tabella 2.2: Tabella che mostra come un determinato errore di un analizzatore potrebbe corrispondere a più forme normalizzate

2.5.3 Aggregazione dei report

Una volta definite le trasformazioni da effettuare possiamo introdurre l'effettivo argomento di questa sezione, cioè l'aggregazione dei quattro file prodotti dagli analizzatori. Il processo di aggregazione permette di generare un unico report finale degli errori, andando a prendere soltanto quelli che sono stati individuati da almeno n analizzatori. Modificando il parametro n andremo, di conseguenza, a modificare la precisione e la dimensione del dataset nel seguente modo:

- Ponendo $n = 1$ avremo la dimensione massima del dataset, in cui ogni singolo errore riportato viene mantenuto, a scapito però di un numero di falsi positivi più grande. Notiamo comunque, e questo vale per tutti i valori di n, che nel caso di duplicati ne viene sempre inserito solo uno.
- Ponendo $n = 2$ avremo un bilanciamento fra precisione e dimensione del dataset.
- Ponendo $n > 2$ invece il numero di errori diventa così basso che renderebbe difficile addestrare una rete, il numero però di falsi positivi diminuisce di conseguenza.

Si può vedere in modo più chiaro come al variare del valore di n cambi il numero di errori ottenuti in Figura 2.3. Nel caso di questo lavoro sono stati utilizzati dataset sia derivanti dal porre $n = 1$ sia dal porre $n = 2$.

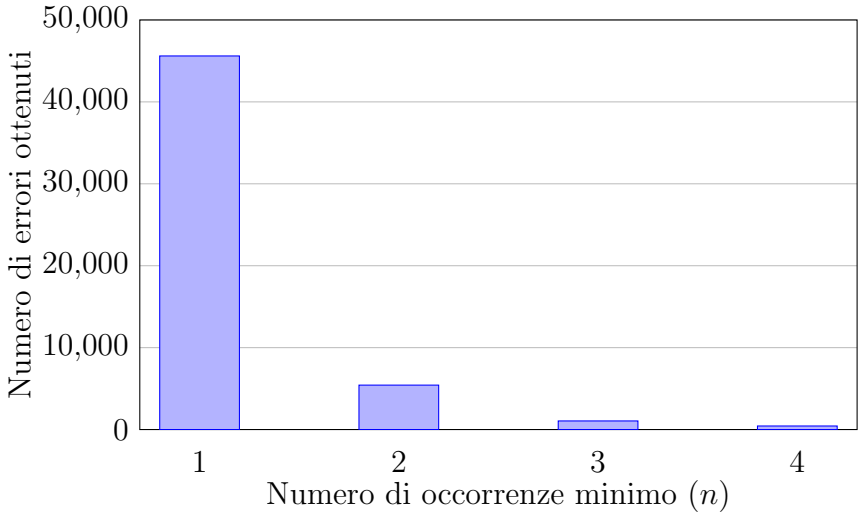

Figura 2.3: Numero di errori aggregati ottenuti in variazione del numero n di occorrenze minime

2.6 Terza fase: associazione tra errore e codice

Lo scopo di questa fase è quello di mappare la posizione di ogni singolo errore ad'un determinato *code snippet*. Prima di far ciò va definito però a che livello eseguire le analisi e quindi le successive predizioni del modello. Le possibili strade che si possono intraprendere possono essere:

- A livello di file. Facendo ciò dato un errore il *code snippet* che associamo è il codice sorgente del intero file. Fare ciò ha due vantaggi principali: la semplicità e la quantità d'informazioni codificate. Ha però anche una serie di svantaggi: per il modello potrebbe essere troppo dispersivo per file grandi e dal momento che un singolo file è probabile che contenga più errori il modello dovrebbe restituire sequenze di predizioni.

- A livello di funzione. In questo caso si associa all'errore il blocco della funzione che lo racchiude. Il beneficio di ciò è la riduzione drastica del frammento di codice associato ad'un errore, rendendo più chiare le relazioni tra i vari elementi del codice e il tipo di errore.

- A livello di riga, in cui ad'un dato errore associamo come code snippet solo la riga stessa. In questo caso la dispersione sarà minima ma allo stesso tempo lo sarà la quantità d'informazioni a disposizione.

In Figura 2.4 possiamo notare il *trade-off* che avviene tra l'aumentare della dimensione del code snippet e la quantità d'informazioni da esso esprimibile.
Nel lavoro svolto è stato scelto di eseguire analisi a livello di funzione, andando però ad aumentare il quantitativo d'informazioni a disposizione aggiungendo un contesto della funzione stessa.

Figura 2.4: *Trade-off* che avviene tra la dispersività e la quantità di informazioni. Selezionando tanto codice avremo tante informazioni ma anche la dispersività aumentata, mentre selezionandone poco avremo poca dispersività ma potremmo perdere informazioni chiave. Le due linee rosse indicano un punto di bilanciamento tra i due.

2.6.1 Alberi di sintassi astratta

Parsing degli AST

2.6.2 Estrazione del codice della funzione

2.6.3 Estrazione del contesto della funzione

Riferimenti a variabile e funzioni esterne

Riferimenti a tipi esterni

2.7 Quarta fase: Generazione degli AST-context

2.7.1 AstMiner

2.7.2 Generazioni vocabolari per i token e per i path

Capitolo 3

Modello vero e proprio

3.1 Soluzioni allo sbilanciamento del dataset

3.1.1 Oversampling

3.1.2 Loss pesata

3.2 Differenti architetture del modello provate

3.3 Training

3.3.1 Metriche utilizzate

3.4 Risultati

3.4.1 Risultati dati dal test dataset

3.4.2 Risultati dati su codice creato al momento

Capitolo 4

Conclusioni

4.1 Possibili migliorie

Bibliografia

[1] Ben Gelman, Banjo Obayomi, Jessica Moore, and David Slater. Source code analysis dataset. *Data in brief*, 27:104712, 2019.

[2] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. User-perceived source code quality estimation based on static analysis metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 100–107. IEEE, 2016.