

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Triennale in Informatica

**Tecniche di deep learning
per il riconoscimento
di errori nel codice**

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Matteo Vannucchi

Sessione I
Anno Accademico 2021-2022

Abstract

Il ruolo dell'informatica è ormai diventato chiave del funzionamento del mondo moderno in progressiva digitalizzazione di ogni singolo aspetto della vita dell'individuo. 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 così 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'obiettivo di esplorare e sviluppare un modello per il riconoscimento di errori in un qualsiasi file sorgente scritto in linguaggio sia C sia C++.

Indice

1	Introduzione teorica	6
1.1	Code2Vec	6
1.1.1	Meccanismo di attenzione	6
1.1.2	AstContext	6
2	Dataset	7
2.1	Dataset originale	7
2.2	Analizzatori di codice statici	8
2.2.1	Analisi a livello di progetto	9
2.2.2	Analizzatori utilizzati	9
2.3	Utilizzo efficace di processori multicore	10
2.4	Prima fase: generazione dei report degli errori	10
2.5	Seconda fase: aggregazione dei report degli errori	12
2.5.1	Parsing dei report	12
2.5.2	Normalizzazione	13
2.5.3	Aggregazione dei report	14
2.6	Terza fase: associazione tra errore e codice	14
2.6.1	Alberi di sintassi astratta	17
2.6.2	Generazione e parsing degli AST	19
2.6.3	Contesto di una funzione	19
2.6.4	Esempio di risultato finale	21
2.7	Quarta fase: Generazione degli AST-context	22

2.7.1	AstMiner	22
2.7.2	Generazioni vocabolari per i token e per i path	23
2.8	Risultato finale della generazione	25
2.8.1	Estrazione di frammenti di codice senza errori	25
3	Il modello predittivo	26
3.1	Struttura	26
3.1.1	Struttura degli input	28
3.1.2	Classificazione	29
3.1.3	Predizione di regressione	30
3.1.4	Reshape degli input	30
3.1.5	Ulteriori architetture del modello provate	30
3.2	Soluzioni allo sbilanciamento del dataset	30
3.2.1	Oversampling	30
3.2.2	Loss pesata	30
3.3	Training	30
3.3.1	Metriche utilizzate	30
3.4	Risultati	30
3.4.1	Risultati dati dal test dataset	30
3.4.2	Risultati dati su codice creato al momento	30
4	Conclusioni	31
4.1	Possibili migliorie	31
	Bibliografia	32

Elenco delle figure

1	Struttura del modello	5
2	Struttura del modello utilizzato	5
2.1	La struttura della directory di un progetto del dataset iniziale	8
2.2	Numero di errori generati da ogni analizzatore	12
2.3	Numero di errori aggregati ottenuti in variazione del numero n di occorrenze minime	15
2.4	<i>Trade-off</i> che avviene tra la dispersività e la quantità di informazioni. Selezionando tanto codice avremo molte informazioni codificate ma aumenterà, allo stesso tempo, la dispersività, mentre selezionandone poco avremo poca dispersività ma potremmo perdere informazioni chiave. Le due linee rosse indicano un punto di bilanciamento tra i due.	16
2.5	Esempio di albero di sintassi astratta	18
2.6	Grafo delle dipendenze di tipo per lo Snippet 2.3	21
2.7	Dimensione dei vocabolari dei token	24
3.1	Struttura astratta del modello utilizzato	27
3.2	Struttura del modello utilizzato	27

Elenco delle tabelle

2.1	Tabella delle diverse nomenclature per l'errore 'memory leak'	13
2.2	Tabella che mostra come un determinato errore di un analizzatore potrebbe corrispondere a più forme normalizzate	13

Introduzione

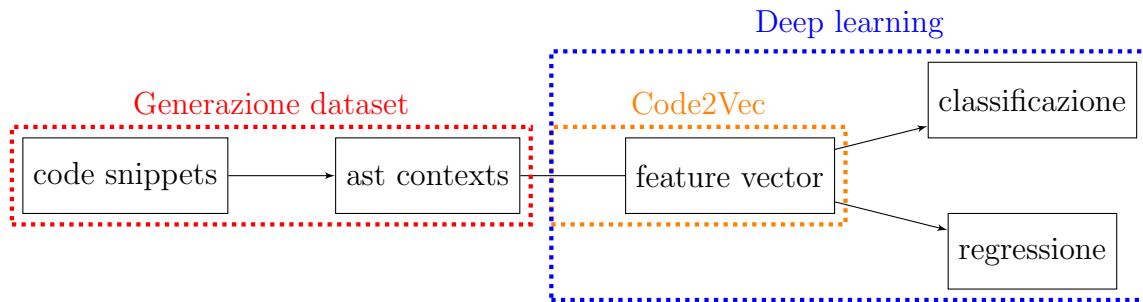


Figura 1: Struttura del modello

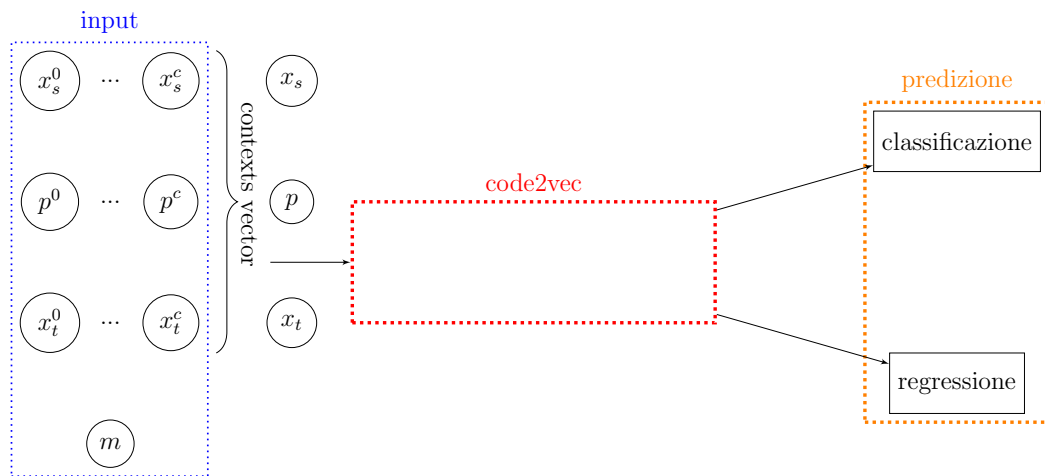


Figura 2: Struttura del modello utilizzato

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. In prima istanza vedremo la versione originale utilizzata e poi come è stata migliorata tramite l'utilizzo di ulteriori analizzatori per aumentarne la precisione delle rilevazioni, riducendone il numero di falsi positivi. In un secondo momento verrà esposto come le rilevazioni degli analizzatori statici sono utilizzate per la associazione tra un *code snippet* e il relativo errore, infine 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 [2]. Il dataset è composto da 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 serve per garantire che i progetti all'interno del dataset soddisfino degli standard di qualità, infatti come precedenti studi hanno mostrato (come ad esempio [4]) 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 Do-

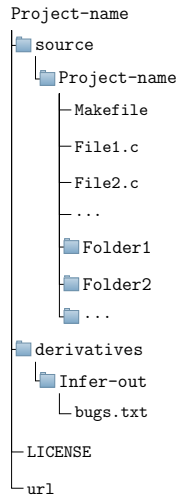


Figura 2.1: La struttura della directory di un progetto del dataset iniziale

xygen che estrae le coppie codice-commento e l'analisi di Infer ¹ che produce un report di analisi statica degli errori. L'analisi di Doxygen è stata scartata, in quanto non utile allo scopo di questo lavoro. In Figura 2.1 si può vedere la struttura tipica di uno dei circa 3000 progetti presenti. Come si può notare, ogni directory contiene un Makefile necessario per l'esecuzione corretta di alcuni analizzatori.

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>, spesso in forma di file testuale. 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.

¹Infer è un analizzatore di codice statico

Gli analizzatori non sono però perfetti, infatti nell'insieme degli errori trovati si possono spesso trovare dei falsi positivi: frammenti di codice segnalati come erronei che in realtà non presentano nessun tipo di problema.

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 a risolvere correttamente gli *include* (nel caso di C e C++) e generando un output più significativo. Alcuni di questi, per far ciò, hanno bisogno di un file 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 utilizzati

Come analizzatori sono stati utilizzati i seguenti quattro:

- L'analizzatore Cppcheck che ha tra i suoi punti di forza il minimizzare il numero di falsi positivi.
- GCC che, oltre ad'essere un compilatore, ha anche funzionalità per l'analisi statica dei programmi attraverso il parametro *-fanalyzer*.
- Il compilatore Clang, che attraverso un suo tool chiamato Clang-Check, è in grado di effettuare analisi statiche.
- L'analizzatore Infer il cui dataset è già dotato delle analisi.

Non sono stati usati analizzatori dinamici, questo perché il loro utilizzo in modo automatizzato è un'operazione estremamente complicata e al di fuori della portata di questo lavoro; infatti quasi tutti i programmi prendono dei parametri o degli input durante l'esecuzione, ma fornire questi dati in modo consistente, sensato per il programma e in modo automatizzato è praticamente impossibile.

L'utilizzo di essi potrebbe però portare a risultati molto interessanti poiché parte dei falsi positivi degli analizzatori statici deriva dal non poter decidere se frammenti di codice sono o non sono eseguiti e di conseguenza devono analizzarli tutti. Può infatti succedere che l'analizzatore statico riferisca errori presenti in codice mai eseguito mentre, in questo caso, quello dinamico, correttamente, non lo riferirebbe.

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 potrebbero estendersi a periodi di più giorni. Dal momento che il processore a disposizione è *multicore*, è stato deciso di ridurre i tempi di esecuzione delle fasi della generazione sfruttando appieno questa caratteristica. 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 modalità.

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 con estensione '.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 contenente 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 occupandosi lui di trovare i file da analizzare. Non viene però utilizzato in questa modalità per una motivazione principale: al posto di utilizzare il percorso assoluto o relativo di un file, clang-check utilizza solamente il nome di questo. Può succedere però che in grandi progetti si abbiano file chiamati uguali ma in cartelle diverse e quindi la loro distinzione sarebbe impossibile. Per risolvere questo problema viene eseguito individualmente su ogni file tramite il comando:

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

Gli output generati dall'esecuzione di questi comandi vengono poi processati andando a sostituire i nomi dei file con il loro percorsi relativi, infine sono uniti tutti insieme. Come si può notare viene utilizzato 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.

- 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.

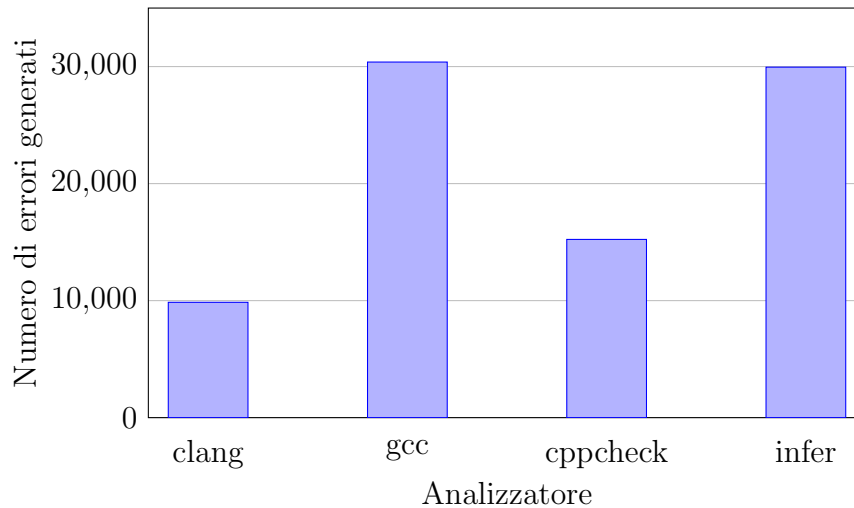


Figura 2.2: Numero di errori generati da ogni analizzatore

2.5 Seconda fase: aggregazione dei report degli errori

Dopo la prima fase, descritta nella sezione precedente, avremo come risultato quattro report di errori per ogni progetto 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 $\langle \text{errore}, \text{posizione} \rangle$, dove la posizione indica sia il percorso del file sia 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, memlea- kOnRealloc, ...	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'.

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

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.

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 selezionare soltanto gli errori 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 errori duplicati ne viene sempre inserito solo uno.
- Ponendo $n = 2$ avremo un bilanciamento fra precisione e dimensione del dataset. Vengono infatti selezionati tutti gli errori riferiti da almeno due analizzatori.
- Ponendo $n > 2$ invece il numero di errori selezionato diventa così basso che renderebbe difficile addestrare il modello, il numero di falsi positivi però 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 $n = 2$.

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

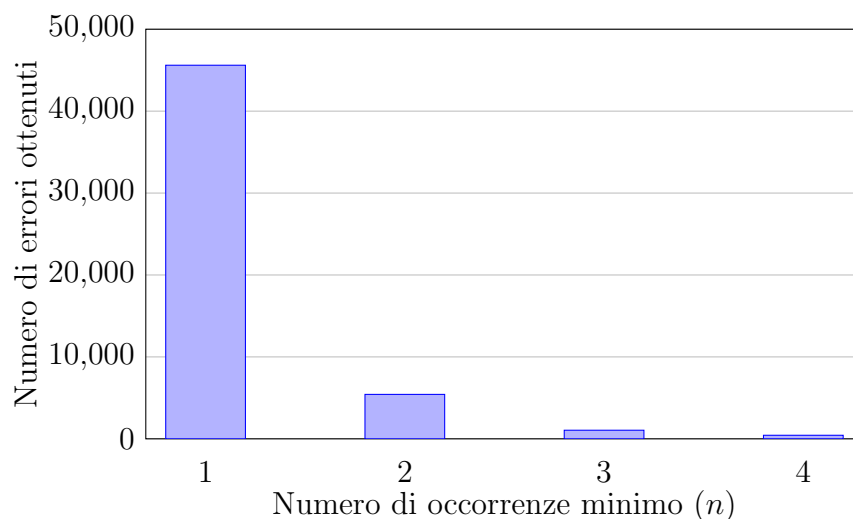


Figura 2.3: Numero di errori aggregati ottenuti in variazione del numero n di occorrenze minime

analisi e quindi le successive predizioni del modello. Le possibili strade che si possono intraprendere sono:

- A livello di file. Facendo ciò, dato un errore, il *code snippet* che associamo è il codice sorgente del intero file. Questo metodo 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, 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'aumento della dimensione del code snippet e la quantità d'informazioni da esso incapsulata.

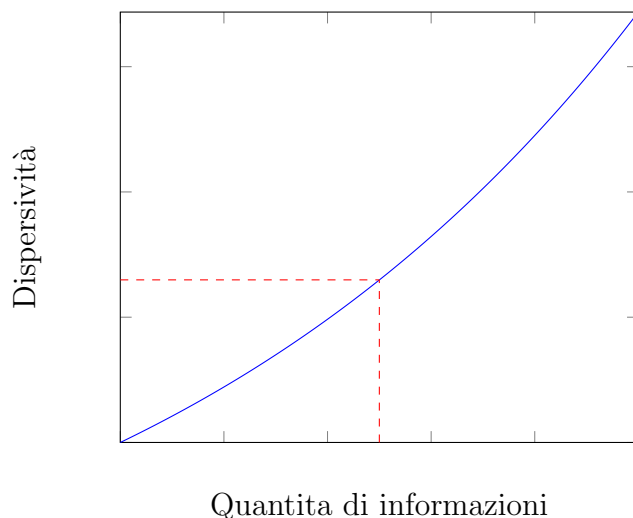


Figura 2.4: *Trade-off* che avviene tra la dispersività e la quantità di informazioni. Selezionando tanto codice avremo molte informazioni codificate ma aumenterà, allo stesso tempo, la dispersività, mentre selezionandone poco avremo poca dispersività ma potremmo perdere informazioni chiave. Le due linee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

Nel lavoro svolto è stato scelto di eseguire le analisi a livello di funzione, andando però ad aumentare il quantitativo d'informazioni a disposizione aggiungendo un contesto della funzione (la cui definizione verrà data in seguito nella sottosezione 2.6.3). Facendo questa scelta, è possibile approssimare il problema ipotizzando che in una data funzione ci sia massimo un solo errore, rendendo quindi l'architettura del modello finale più semplice. Una volta determinato il livello a cui svolgere le analisi possiamo tornare al problema principale: estrarre il codice della funzione che racchiude l'errore. Nonostante questo possa sembrare un problema semplice di analisi testuale vedremo come, in realtà, non lo sia. Questo vale ancora di più per linguaggi come C e C++ che, tramite la loro sintassi molto libera, rendono il tutto più complicato. Prima di discutere di ciò dobbiamo quindi introdurre un concetto fondamentale: gli alberi di sintassi astratta.

2.6.1 Alberi di sintassi astratta

Un albero di sintassi astratta, in breve *ast* (dall'inglese *abstract syntax tree*), è una rappresentazione ad albero della struttura sintattica astratta di un testo, nel nostro caso del codice sorgente. Queste strutture sono spesso generate da parser specifici e vengono utilizzati come rappresentazione intermedia del programma in un processo di compilazione.

Possiamo vedere un esempio di albero prendendo il frammento di codice nello Snippet 2.1 scritto in uno pseudo-linguaggio. Un possibile albero di sintassi astratta per questo frammento di codice è quello indicato in Figura 2.5. Come si può vedere l'*ast* rappresenta in modo efficace la struttura completa del codice, andando ad individuare e distinguere elementi come:

- Il corpo, la *signature* e il valore di ritorno della funzione.
- La dichiarazione della variabile 'total' e il blocco del *for*.
- I componenti del costrutto *for*: il corpo, la condizione, l'inizializzazione e lo step della variabile di controllo.

Snippet 2.1: Frammento di codice che calcola la somma dei valori in un vettore

```
1 void foo(int[] array){
2     int totale = 0;
3     for(int i = 0; i < array.length; i++){
4         totale = totale + array[i]
5     }
6     return totale;
7 }
```

In questo progetto verranno utilizzati gli alberi di sintassi astratta per poter estrarre correttamente il codice della funzione che racchiude l'errore. Questa operazione poteva anche essere fatta a livello testuale ricercando le parentesi graffe di apertura e chiusura della funzione. In questa maniera, però, sorgono casi problematici come l'utilizzo di

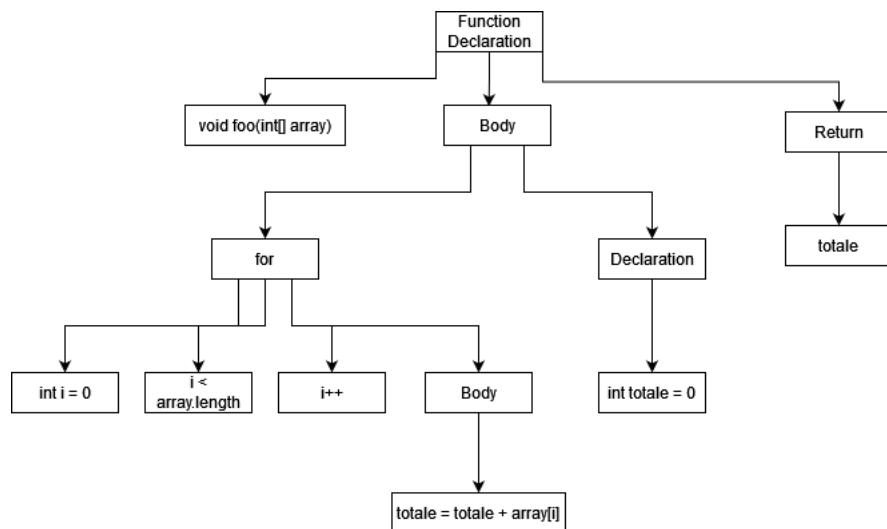


Figura 2.5: Esempio di albero di sintassi astratta

parentesi graffe all'interno di commenti (o stringhe) che vanno ignorati o l'utilizzo particolare di direttive *define*. Tutto ciò rende il problema, in apparenza semplice, in realtà complesso. A supporto di ciò vediamo lo Snippet 2.2 in cui, nonostante sia codice C++ completamente valido, l'individuazione del corpo della funzione che racchiude la linea di errore è un'operazione tutt'altro che semplice.

Snippet 2.2: Esempio di codice valido con struttura particolare

```

1  #DEFINE OPENBRACKET {
2  #DEFINE CLOSEBRACKET }
3  void foo() OPENBRACKET
4      int error = 5 / 0; // Linea contenente l'errore
5
6      /* Questo commento rende difficile l'individuazione del
7      corpo della funzione */
8
9  CLOSEBRACKET

```

2.6.2 Generazione e parsing degli AST

Come accennato nella precedente sezione, in questo lavoro vengono utilizzati gli *ast* come metodo per estrarre i blocchi delle funzioni e il loro relativo contesto. Prima di tutto bisogna essere in grado di generare un albero di sintassi astratta dato un file sorgente. Per far ciò viene utilizzato il compilatore Clang che, attraverso flag specifiche, è in grado di generare un albero rappresentato in formato JSON. Più in particolare per ogni file sorgente che vogliamo analizzare viene eseguito il seguente comando:

```
$ clang -Xclang -ast-dump=json <file>
```

Il risultato di questa operazione è un file in formato JSON che rappresenta la struttura dell'albero. Prima però di proseguire, questo file viene caricato e trasformato in una struttura ad albero vera e propria.

Oltre alla struttura sintattica del codice, all'interno dei dizionari JSON, sono presenti informazioni ulteriori: indicazioni sulla posizione degli elementi, sui tipi, sui riferimenti esterni e molto altro. Tutte queste informazioni vengono poi usate sia per estrarre il codice sia per estrarre il contesto della funzione.

2.6.3 Contesto di una funzione

Definiamo in fine cosa si intende con contesto di una funzione. Il contesto di una funzione è l'insieme di tutti quei riferimenti esterni che vengono effettuati all'interno del corpo della funzione, possono includere:

- Funzioni esterne.
- Variabili esterne.
- Definizioni di tipo esterne. Visto che le definizioni di tipo possono dipendere da altri tipi non primitivi, in questo caso oltre al riferimento stesso vengono aggiunte anche le dipendenze di esso, dove per dipendenza di un tipo t al tipo v si intende che la dichiarazione di t include il tipo v (un esempio di ciò lo si può vedere nello Snippet 2.3).

L'idea posta alla base dell'includere questo contesto nel risultato finale è il poter dare al modello il maggior numero d'informazioni possibili mantenendo comunque limitata la dimensione dello snippet. Senza di questo, infatti, anche un umano potrebbe non essere in grado di comprendere il codice o non poterne trarre conclusioni significative su di esso. Guardando infatti lo Snippet 2.3, senza l'inclusione nel risultato finale della variabile globale non potremmo determinare che nella funzione *foo* ci sia effettivamente un errore di divisione per zero.

Snippet 2.3: Esempio di codice con riferimenti esterni

```
1
2     typedef int typeA;
3     typedef typeA typeB;
4
5     typedef float typeC;
6
7     int variabileGlobale = 0;
8
9     int bar(){
10         ...
11     }
12
13     void foo(){
14         typeB variable = 1; //riferimento di tipo esterno
15         int var = 5 / variabileGlobale // errore
16
17         result = bar();
18         ...
19     }
```

Estrazione del contesto

Per l'estrazione del contesto vengono utilizzate le informazioni codificate nell'albero di sintassi astratta prodotto da Clang. Solamente per le definizioni di tipo vengono incluse anche le relative dipendenze e per far ciò viene costruito un grafo delle dipendenze.

Il grafo delle dipendenze è un grafo diretto in cui un vertice v rappresenta una dichiarazione di tipo, mentre un arco (u, v) rappresenta la dipendenza del tipo u dal tipo v . Prendiamo come esempio lo Snippet 2.3. Il contesto della funzione *foo* in questo caso dovrà mantenere informazioni sulla definizione di *typeB*, dipendente però dalla definizione del *typeA*. Bisogna quindi costruire il grafo delle dipendenze dei tipi utilizzando le informazioni contenute all'interno dell'albero di sintassi astratta. Possiamo vedere quindi il grafo delle dipendenze per questo specifico frammento di codice in Figura 2.6.

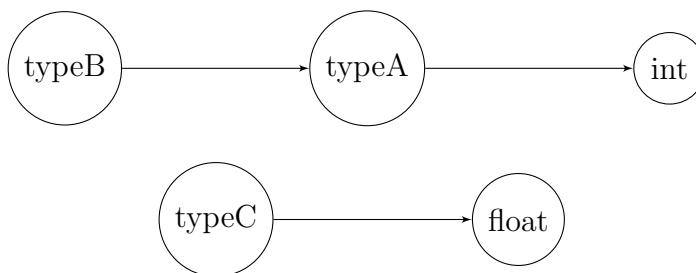


Figura 2.6: Grafo delle dipendenze di tipo per lo Snippet 2.3

Per poter ricavare le dipendenze delle dichiarazioni di tipo, una volta costruito il grafo, si può eseguire una visita in ampiezza partendo dal riferimento esterno stesso. Nel caso della funzione *foo* otterremo quindi, visto il riferimento esterno al tipo *typeB*, due definizioni di tipo, quelle di *typeB* e *typeA*.

2.6.4 Esempio di risultato finale

Riprendendo sempre lo Snippet 2.3 avremo che il risultato finale dell'estrazione del metodo *foo* è il seguente:

Snippet 2.4: Esempio di estrazione del codice della funzione *foo* insieme al contesto

```

1
2     typedef int typeA;
3     typedef typeA typeB;
4     int variabileGlobale = 0;
5     int bar();
6     void foo(){
7         typeB variable = 1; //riferimento di tipo esterno
8         int var = 5 / variabileGlobale // errore
9         ...
10    }

```

Notiamo quindi che la definizione del tipo *typeC* non è inclusa poiché non utilizzata all'interno del corpo della funzione, mentre, sia la variabile globale, sia le due definizioni di tipo e sia la signature della funzione *bar* sono incluse nel contesto.

2.7 Quarta fase: Generazione degli AST-context

Come accennato nell'introduzione teorica nel Capitolo 1, il modello che verrà utilizzato prenderà in input il codice sorgente processato sotto forma di vettore di *ast contexts*, cioè delle triple della forma:

$$\langle x_s, p, x_t \rangle$$

dove x_s e x_t sono rispettivamente il *token start* e *token end*, mentre p è il *path* come descritto in [1]. A differenza però di come viene illustrato nell'articolo, in cui x_s e x_t sono un singolo valore, verranno utilizzati dei vettori di token di inizio e fine, nel tentativo di ridurre la dimensione dei vocabolari. Per generare queste triple verrà utilizzato un tool chiamato *Astminer*.

2.7.1 AstMiner

Astminer rappresenta il lavoro descritto in [3] ed è un tool che permette di estrarre gli ast context da file sorgenti scritti in vari linguaggi come Python, C/C++ e Java. Può

essere utilizzato in due modi differenti:

- Come una libreria di Kotlin/Java.
- Come un tool standalone della CLI. Questo sarà il modo che verrà utilizzato nel progetto essendo stato scritto tutto in Python.

Lo strumento è configurabile in svariati modi; le uniche configurazioni rilevanti utilizzate sono:

- É stato utilizzato in modalità code2vec.
- Sono stati utilizzati i seguenti valori:
 - `maxPathContextsPerEntity = 200`. Questo valore rappresenta il numero massimo di *ast context* da associare ad'un frammento di codice.
 - `maxPathLength = 20`. Questo valore rappresenta, invece, la lunghezza massima dei cammini.
 - `nodesToNumbers = false`. Infatti i vocabolari per la traduzione dei token verranno gestiti, non da Astminer, ma a livello dell'applicazione.

Come però già accennato nell'introduzione di questa sezione, il prodotto di Astminer in realtà non è esattamente uguale a quello illustrato nell'articolo di code2vec: invece che avere dei singoli valori per x_s e x_t , Astminer produce in output *ast context* che hanno token d'inizio e fine che sono vettori, scomponendo token complessi in token multipli. É stato deciso di non modificare questa scelta poiché potrebbe portare ad una riduzione notevole della dimensione del vocabolario dei token, rendendo più significativo ogni singolo token.

2.7.2 Generazioni vocabolari per i token e per i path

L'output dell'esecuzione di astminer, visto il parametro `nodesToNumbers = false`, sono degli *ast context* dove ogni token è ancora in formato letterale. Per essere utilizzabile

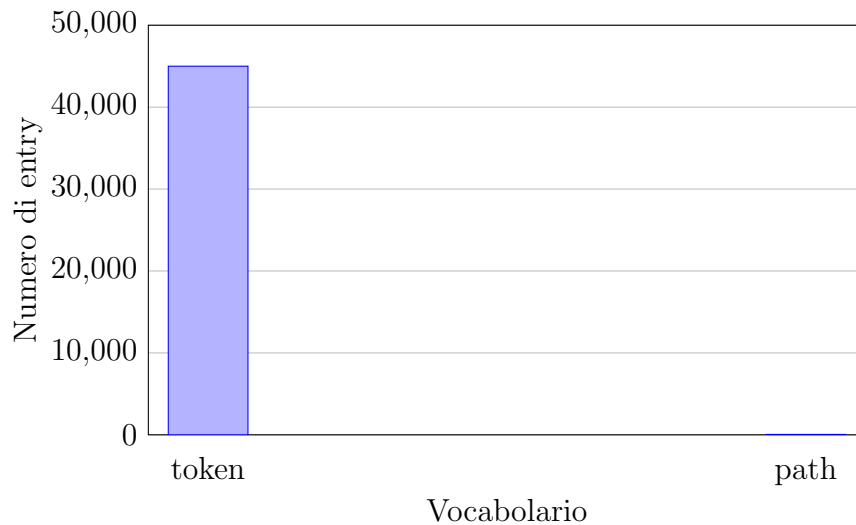


Figura 2.7: Dimensione dei vocabolari dei token

questo formato deve però prima essere trasformato in un valore intero. Tale valore rappresenterà l'indice del token letterale all'interno di uno specifico vocabolario.

Più in particolare vengono creati due dizionari diversi:

- Un dizionario dei token degli elementi dei cammini (p) che d'ora in avanti chiameremo *path vocab*.
- Un dizionario dei token degli elementi terminali (i valori x_s e x_t del *ast context*). In questo caso lo chiameremo *token vocab*.

In Figura 2.7 possiamo vedere la dimensione dei due vocabolari. Si può vedere come i due abbiano ordini di grandezza completamente differenti, questo può essere spiegato dal fatto che i token terminali racchiudono svariate tipologie di elementi come nomi di variabili e di metodi, mentre i token dei cammini racchiudono solamente elementi sintattici fissi: costrutti come dichiarazioni, blocchi e operazioni.

2.8 Risultato finale della generazione

Il risultato dell'esecuzione di tutte le fasi menzionate è quindi un dataset formato da due elementi chiave:

- I *code snippet* sotto forma di vettori di *ast contexts* a cui vengono associate due informazioni:
 - Un'etichetta indicante la tipologia di errore o la classificazione di una funzione senza errori.
 - Il numero della riga dell'errore.
- I due vocabolari dei token dei cammini e dei valori terminali.

2.8.1 Estrazione di frammenti di codice senza errori

Non è ancora stato menzionato come vengono ricavati i *code snippet* che non presentano errori. Il modello finale, infatti, dovrà essere in grado di determinare sia la tipologia di errore sia se effettivamente il frammento presenta errori. Questa estrazione avviene in modo molto semplice: tutte le funzioni che non presentano errori sono estratte.

Come però vedremo nel Capitolo 3, il dataset è molto sbilanciato verso la classe di non errore. Per ridurre ciò, e anche per rendere il file del dataset più maneggiabile, vengono solo analizzati file che presentano errori. Di conseguenza avremo che il numero di funzioni senza errori viene ridotto drasticamente.

Capitolo 3

Il modello predittivo

Nel seguente capitolo affronteremo lo sviluppo del modello predittivo. Vedremo, prima di tutto, la struttura del modello discutendone i principali componenti e varie iterazioni di essa. In un secondo momento vedremo un problema fondamentale dato dalla distribuzione del dataset: il problema dello sbilanciamento. Verrà anche introdotto brevemente come viene addestrato e le metriche utilizzato per valutarlo. Infine saranno discussi i risultati ottenuti.

3.1 Struttura

Come già introdotto nel Capitolo 1, questo modello si basa su un meccanismo di codifica del codice separato in due fasi:

- La prima codifica del *code snippet* in un vettore di *ast contexts*, effettuata a tempo di creazione del dataset, come già discusso nel Capitolo 2.
- La seconda codifica del vettore di *ast contexts* in un vettore di *feature* attraverso meccanismi di *Deep Learning*.

Una volta ottenuto il vettore delle feature, vengono utilizzati due 'sotto reti' per la classificazione e la regressione. Possiamo vedere riassunta a grandi linee la struttura della rete in Figura 3.2.

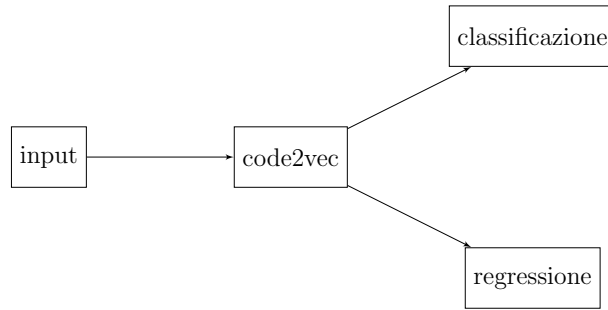


Figura 3.1: Struttura astratta del modello utilizzato

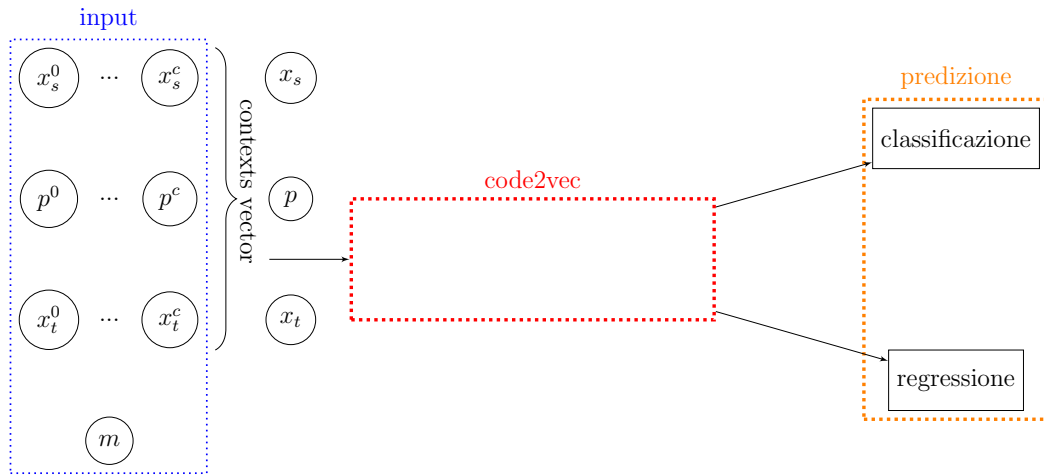


Figura 3.2: Struttura del modello utilizzato

Nelle successive sezione, in maniera approfondita, le seguenti tematiche:

- La struttura degli input e come sono stati gestiti i cambiamenti della loro forma discussi in precedenza nel Capitolo 2.
- La struttura del modello di classificazione.
- La struttura del modello di regressione.

3.1.1 Struttura degli input

Il dataset generato nel Capitolo 2 contiene per ogni suo elemento un vettore di *ast contexts*, cioè un vettore di triple della forma:

$$(x_s, p, x_t)$$

tali per cui vale la seguente relazione:

$$x_s, x_t \in \mathbb{N}^l \quad p \in \mathbb{N}^k$$

dove l e k rappresentano rispettivamente la lunghezza massima del vettore dei token di inizio/fine e la lunghezza massima del vettore dei cammini, fissate al momento della creazione del dataset (nel nostro caso $l = 10$ e $k = 20$). Per poter essere inserito all'interno di un modello dobbiamo

Come si può vedere nella Figura 3.2 l'input è formato da quattro vettori diversi:

- La tripla (x_s, p, x_t) che rappresenta l'*ast context*, tale per cui:

$$x_s, x_t \in \mathbb{N}^{c \times l}, \quad p \in \mathbb{N}^{c \times k}$$

dove l e k rappresentano rispettivamente la lunghezza massima del vettore dei token di inizio/fine e la lunghezza massima del vettore dei cammini (fissate al momento della creazione del dataset). , mentre c rappresenta il numero di massimo *ast context* per ogni dato del dataset. Succede spesso che i vettori non siano di lunghezza sufficiente, in questi casi vengono aggiunti degli elementi di *padding* tali da raggiungere la lunghezza di l o k . Questi elementi di *padding* sono rappresentati da un valore particolare nei rispettivi vocabolari.

- Una vettore maschera m di lunghezza k definito nel seguente modo:

$$m_i = \begin{cases} 1 & \text{se } p_i \text{ non è padding} \\ 0 & \text{altrimenti} \end{cases}$$

Non è invece definito un vettore maschera per i vettori dei token d'inizio e fine, questo perché l'implementazione avrebbe aumentato di troppo la dimensione della rete.

Come già accennato

3.1.2 Classificazione

L'obiettivo della classificazione in questo modello è il predire la classe di errore o l'assenza di errore. Il modello, di conseguenza, in output dovrà fornire un vettore c tale per cui per ogni i :

$$0 < c_i < 1$$

avremo quindi che il vettore c è una *distribuzione di probabilità* delle classi da predire. Di conseguenza la classe con maggior probabilità sarà la classe predetta. Più formalmente la classe predetta i sarà:

$$\arg \max_i c_i$$

Nel lavoro svolto la rete di classificazione è costituita da una serie di *hidden dense layer* culminanti in un *layer* di predizione che utilizza come funzione di attivazione la funzione *softmax*.

3.1.3 Predizione di regressione

3.1.4 Reshape degli input

3.1.5 Ulteriori architetture del modello provate

3.2 Soluzioni allo sbilanciamento del dataset

3.2.1 Oversampling

3.2.2 Loss pesata

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] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.
- [2] Ben Gelman, Banjo Obayomi, Jessica Moore, and David Slater. Source code analysis dataset. *Data in brief*, 27:104712, 2019.
- [3] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press, 2019.
- [4] 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.