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 diventato chiave del funzionamento del mondo moderno, ormai sempre più in progressiva digitalizzazione di ogni singolo aspetto della vita dell'individuo. Con l'aumentare della complessità e delle dimensioni dei programmi, il rilevamento di errori diventa sempre di più un'attività difficile e che necessita l'impiego di tempo e risorse. 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 di grandi dimensioni e la percentuale di falsi positivi possono, infatti, diventare un importante 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 l'obbiettivo di esplorare e sviluppare un modello di *Deep Learning* per il riconoscimento di errori in un qualsiasi file sorgente scritto in linguaggio C e C++.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Con l'aumentare della complessità e della portata dei progetti moderni, lo sviluppo di *tools* in grado di rilevare errori nei programmi si è reso sempre più inteso e necessario. Di conseguenza, sono emersi strumenti di analisi moderni, più o meno sofisticati e multifunzionali, per differenti linguaggi di programmazione.

Inizialmente quest'ambito era quasi completamente dominato da strumenti di analisi tradizionali, che fanno utilizzo di tecniche come *data-flow analysis* [5] e *symbolic execution* [7]. Questa tipologia di strumenti risente, però, di un numero di falsi positivi, cioè rilevazioni di errori non esistenti, piuttosto alto. Di conseguenza, in parallelo ai progressi avuti in ambito di intelligenza artificiale, sono stati esplorati meccanismi di analisi nuovi, tramite tecniche di *Machine Learning* e *Deep Learning*. Queste hanno il potenziale sia di ridurre il problema dei falsi positivi, sia di diminuire altri problemi come il tempo di esecuzione dell'analizzatore e la portabilità fra linguaggi diversi.

In letteratura, sono numerosi gli studi effettuati. Nella ricerca [1], per esempio, l'autore effettua un'indagine sulle diverse tecniche di *Deep Learning* per la predizione di difetti nel codice, presentando le principali difficoltà (come la mancanza di dati e la complessità del contesto del codice) e delle possibili soluzioni per superarle parzialmente. Viene sottolineata, però, la necessità di condurre ancora ulteriori studi per ottenere risultati accettabili. Similmente, nel lavoro [11], viene presentato uno studio su diverse metodologie di *Machine Learning* per l'analisi del codice, includendo svariate tipologie di *task* (come l'analisi della qualità del codice, della rappresentazione di esso e, di interesse a questo lavoro, l'individuazione di errori).

In questo lavoro, verrà generato un dataset con un numero ridotto di falsi positivi. In

seguito, addestreremo un modello di *Deep Learning*, basato sull'architettura Code2Vec (definita nella ricerca [2]), sul dataset generato e ne valuteremo i risultati. In particolare, il lavoro di tesi sarà suddiviso nel seguente modo:

- Nel capitolo 1 vedremo un'introduzione al modello Code2Vec.
- Nel capitolo 2 discuteremo della generazione del dataset alla base dell'addestramento del modello.
- Nel capitolo 3 illustreremo come il modello Code2Vec è stato adattato allo scopo di predire gli errori e i risultati ottenuti.

Capitolo 1

Introduzione teorica

Nel seguente capitolo verranno introdotte le basi teoriche utili alla comprensione del presente lavoro. In particolare, vedremo prima gli alberi di sintassi astratta, utilizzati per la generazione del dataset, e in seguito sommariamente il modello Code2Vec, introdotto per la prima volta nella ricerca [2], posto alla base del modello utilizzato che verrà discusso nel Capitolo 3. Fondamenti di *Machine Learning* e *Deep Learning* non saranno invece trattati.

1.1 Albero 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 utilizzate come rappresentazione intermedia del programma in un processo di compilazione. In maniera più formale, un albero di sintassi astratta per un frammento di codice C è una tupla del tipo:

$$(N, T, X, s, \delta, \phi)$$

dove N è l'insieme dei nodi non terminali, T l'insieme dei nodi terminali, X un insieme di valori, s la radice, δ la funzione che associa ad un non terminale la lista di nodi figli e ϕ la funzione che associa ad un terminale un valore.

Possiamo vedere un esempio di albero prendendo il frammento di codice nello Snippet 1.1 scritto in uno pseudo-linguaggio. Un possibile albero di sintassi astratta per questo frammento di codice è quello indicato in Figura 1.1. 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* come il corpo, la condizione, l'inizializzazione e lo step della variabile di controllo.

Snippet 1.1: 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  }
```

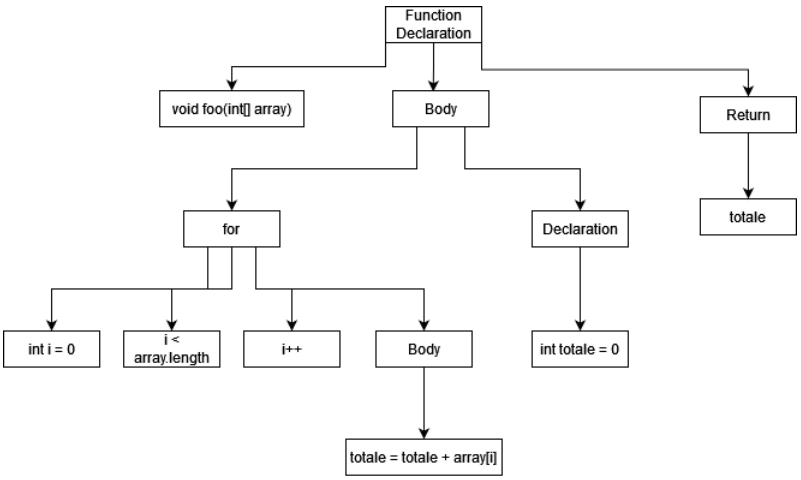

Figura 1.1: Esempio di albero di sintassi astratta

1.2 Code2Vec

Il modello Code2Vec, sviluppato nella ricerca [2], ha come obbiettivo la rappresentazione di un *code snippet*, di lunghezza variabile, come un vettore di lunghezza fissa. Questa operazione di trasformazione viene detta *embedding* e il suo prodotto può essere utilizzato per ulteriori *task* come la predizione del nome del metodo, come studiato nell'articolo, o, come nel caso di questo lavoro, la predizione di errori.

A differenza di altri modelli per la generazione di un *embedding* del codice, questo metodo utilizza tecniche di *Deep Learning* sfruttando principalmente un meccanismo di attenzione. Prima di poter descrivere la struttura del modello, bisogna introdurre il concetto di *ast contexts*.

Definizione 1 (Ast context) *Dato un albero di sintassi astratta, definiamo con ast context le triple della forma:*

$$(x_s, p, x_t)$$

dove p rappresenta un cammino fra due nodi terminali nell'albero, mentre x_s e x_t rappresentano i valori del nodo d'inizio e di fine, cioè $x_s = \phi(p_0)$ e $x_t = \phi(p_k)$, dove k è la lunghezza del cammino.

Per la rappresentazione del codice verrà utilizzato un insieme $B = \{b_1, .., b_c\}$ di *ast context*.

1.2.1 Struttura del modello

In Figura 1.2 possiamo visualizzare la struttura alla base del modello di Code2Vec. Come si può notare, ci sono quattro componenti fondamentali:

- L'input, costituito da un insieme di *context vector* ottenuti tramite l'*embedding* degli *ast contexts* contenuti nell'insieme B.
- Un *dense layer* che riduce le dimensioni e combina i valori di ogni vettore.
- Un meccanismo di attenzione che impara a combinare i multipli *context vector* in un unico vettore denominato *code vector*.
- Uno strato finale di classificazione, componente che verrà modificata all'interno di questo lavoro.

Il meccanismo di attenzione è la chiave del funzionamento di questo modello. Esso, infatti, attraverso dei pesi imparabili, effettua una media pesata dei *context vectors* riducendoli ad un unico vettore. In questo modo il modello è in grado d'imparare quanta importanza (o, per l'appunto, attenzione) dare ad ogni singolo vettore.

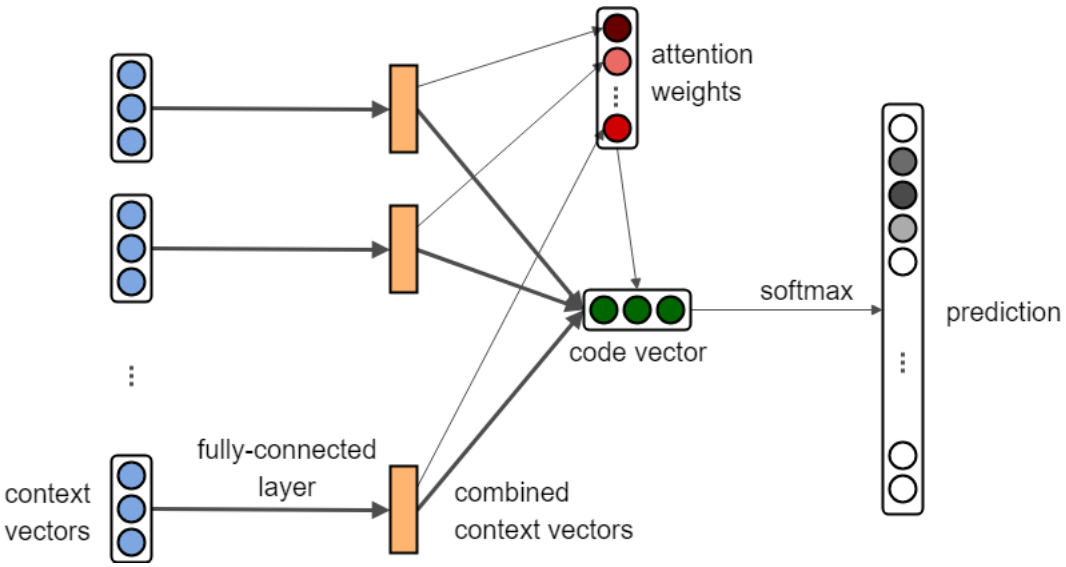

Figura 1.2: Struttura del modello Code2Vec

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 l'associazione tra un *code snippet* e il relativo errore. Infine, si analizzerà 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 generato nella ricerca [6]. 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 [10]), si può utilizzare il numero di stelle su GitHub come un *proxy* per la qualità del codice stesso.

Per ogni progetto, il dataset contiene una serie di analisi effettuate: l'analisi di Do-

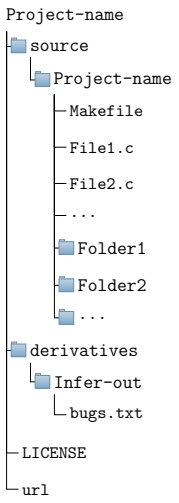

Figura 2.1: La struttura della directory di un progetto del dataset iniziale

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

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.

[1] Infer è un analizzatore di codice statico

Gli analizzatori, però, non sono perfetti, nell'insieme degli errori trovati si possono spesso trovare dei falsi positivi, ovvero 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, 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 analasi.

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.

Utilizzare un analizzatore dinamico piuttosto che un analizzatore statico potrebbe essere interessante, dato che i risultati ottenuti da quest'ultimo contengono più spesso dei falsi positivi. Questo, in parte, dipende dall'impossibilità di decidere se i frammenti di codice sono eseguiti o meno; di conseguenza, essi dovrebbero essere analizzati tutti. Potrebbe succedere che un analizzatore statico riferisca errori presenti in codice mai eseguito: in un caso del genere, invece, l'analizzatore dinamico non riferirebbe la presenza dell'errore.

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 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 può essere, invece, eseguito su una cartella di un progetto, rimuovendo la necessità 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 con lo stesso nome ma in cartelle diverse: in questo caso, la loro distinzione sarebbe impossibile. Per risolvere questo problema Clang-Check 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 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.

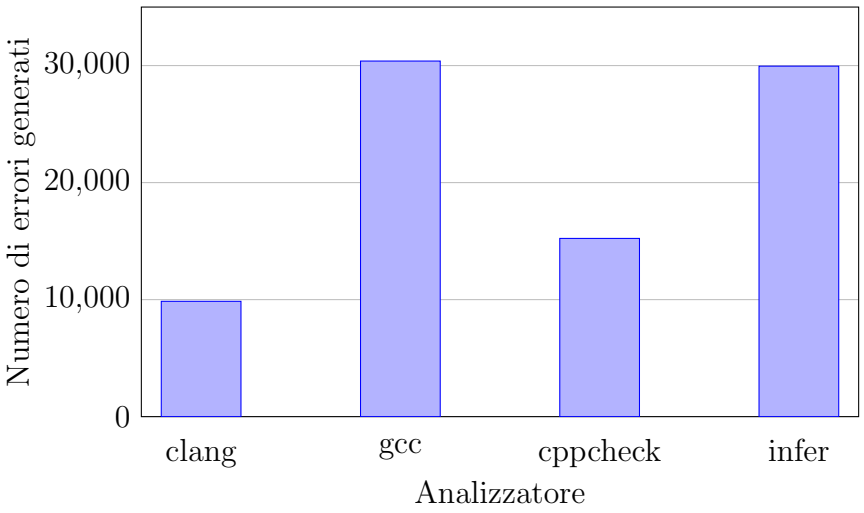

Figura 2.2: Numero di errori generati da ogni analizzatore

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 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 (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. I risultati del parsing sono quindi tanti record nella forma $\langle$*errore, 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. Questa 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 (lo si può vedere nella Tabella 2.2). Nella eventualità 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 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 da rendere difficile l'addestramento del modello. Il numero di falsi positivi, però, diminuisce.

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 derivanti dal porre $n = 2$.

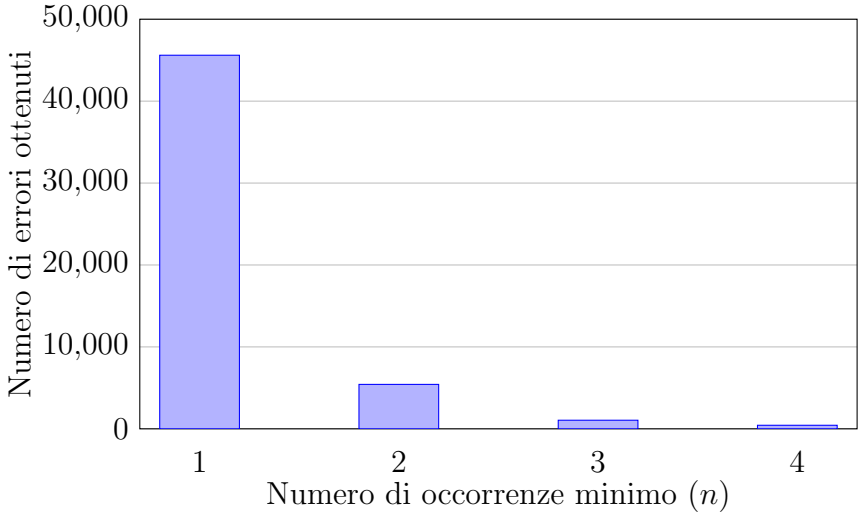

Figura 2.3: Numero di errori aggregati ottenuti al variare del numero n di occorrenze minime

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 a che livello eseguire le 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 dell'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 potrebbe contenere 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 della dimensione del frammento di codice, che rende più chiare le relazioni tra i vari elementi del codice e il tipo di errore.

- A livello di riga. Ad un dato errore associamo come code snippet solo la riga stessa. In questo caso la dispersione sarà minima, ma lo sarà anche la quantità di informazioni.

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.

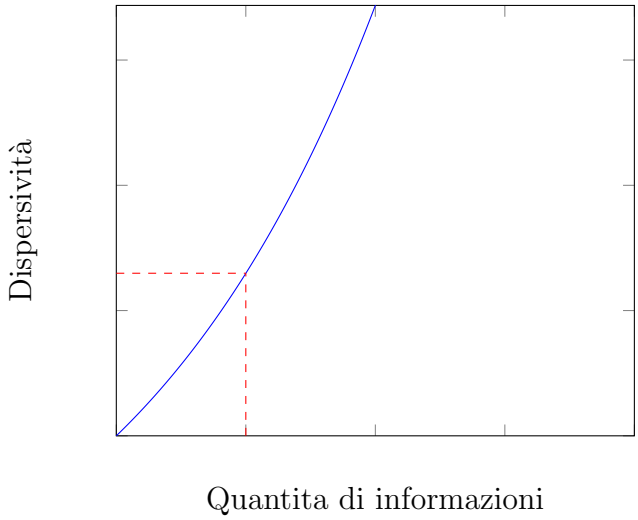

Figura 2.4: *Trade-off* che avviene tra la dispersività e la quantità di informazioni. Selezionando tanto codice avremo molte informazioni codificate ma aumenterà, allo stesso tempo, la dispersività, mentre selezionandone poco avremo poca dispersività ma potremmo perdere informazioni chiave. Le due linee 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 2.6.2). Facendo questa scelta, è possibile approssimare il problema ipotizzando che in una data funzione ci sia al massimo un solo errore, rendendo 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. A supporto di ciò vediamo lo Snippet 2.1.

Snippet 2.1: Esempio di codice C e C++ valido con struttura particolare

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

In questo frammento si possono individuare due fattori problematici: la presenza di parentesi graffe in commenti e l'utilizzo particolare di direttive *define*. Se volessimo ricavare il corpo della funzione analizzando semplicemente il testo, dovremmo trovare le parantesi graffe di apertura e chiusura di esso, ma i due fattori appena elencati e ulteriori non discussi rendono necessarie delle accortezze maggiori.
Per risolvere questo problema utilizzeremo gli alberi di sintassi astratta.

2.6.1 Generazione e parsing degli alberi di sintassi astratta

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. 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.

2.6.2 Contesto di una funzione

Definiamo, infine, 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 2.2).

Includere questo contesto nel risultato finale permette di 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 2.2, 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 2.2: 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    int 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. 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 2.2. 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 in-

formazioni contenute all'interno dell'albero di sintassi astratta. Possiamo vedere quindi il grafo delle dipendenze per questo specifico frammento di codice in Figura 2.5.

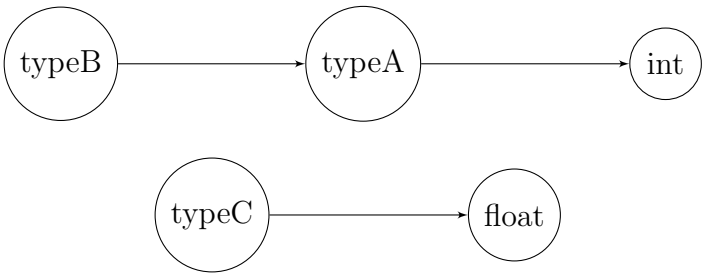

Figura 2.5: Grafo delle dipendenze di tipo per lo Snippet 2.2

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*.

2.6.3 Esempio di risultato finale

Riprendendo sempre lo Snippet 2.2, il risultato finale dell'estrazione del metodo *foo* è rappresentato nello Snippet 2.3.

Snippet 2.3: 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    int result = bar();
11    ...
```

```
12  }
```

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

2.7 Quarta fase: generazione degli ast contexts

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:

$$<x_s, p, x_t>$$

dove x_s e x_t sono rispettivamente il *token start* e *token end*, mentre p è il *path* come descritto in [2]. A differenza 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 nella ricerca [9] ed è un tool che permette di estrarre gli *ast contexts* 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:

- È stato utilizzato in modalità code2vec.

- Sono stati utilizzati i seguenti valori:
  - maxPathContextsPerEntity = 200. Questo valore rappresenta il numero massimo di *ast contexts* 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 non verrano gestiti 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 contexts* che hanno token d'inizio e fine che sono vettori, scomponendo token complessi in token multipli. Si è 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

Gli output dell'esecuzione di Astminer, visto il parametro nodesToNumbers = false, sono degli *ast contexts* dove ogni token è ancora in formato letterale. Per essere utilizzabile, questo formato deve 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 2.6 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

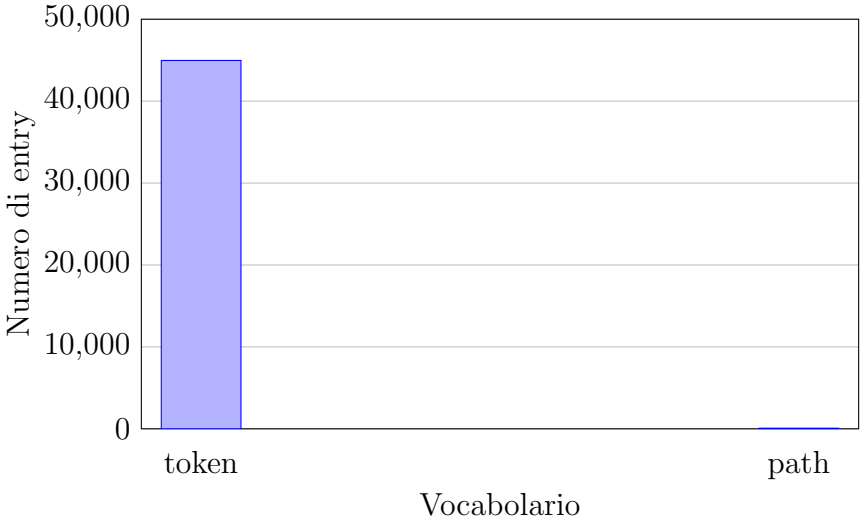

Figura 2.6: Dimensione dei vocabolari dei token

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 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.

In Tabella 2.3 possiamo vedere riassunti sia le tipologie di errori estratti sia la loro frequenza.

Errore	Osservazioni
No Error	41417
Memory leak	1032
Null dereference	911
Dead store	791
Variable not initialized	325
Resource leak	122
Double free	83
Pointer conversion	80
Data not initialized	56
Null argument	31
Use after lifetime	16
Division by zero	10
Integer overflow	4
Call and message	3
Array not initialized	2
Use closed file	1

Tabella 2.3: Tabella riassuntiva degli errori estratti con la loro frequenza

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. Innanzitutto, prenderemo in esame la struttura del modello, discutendone i principali componenti. In un secondo momento, affronteremo un problema dato dalla distribuzione del dataset: il problema dello sbilanciamento. Verrà anche introdotto brevemente come il modello viene addestrato e le metriche utilizzate per valutarlo. Infine, saranno discussi i risultati ottenuti e un'altra possibile struttura di esso.

3.1 Struttura del modello

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 (del tipo di errore) e la regressione (del numero della riga). In Figura 3.1 possiamo vedere riassunta a grandi linee la struttura della rete.

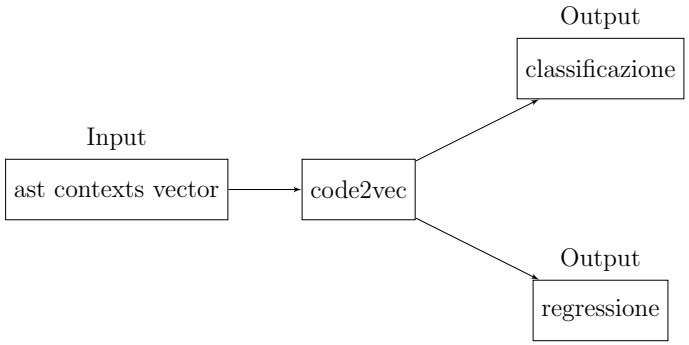

Figura 3.1: Struttura astratta del modello utilizzato

Nelle successive sezioni discuteremo, 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:

$$(x_s^{(i)}, p^{(i)}, x_t^{(i)})$$

tale per cui vale la seguente relazione:

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

dove l e k, fissate al momento della creazione del dataset, rappresentano rispettivamente la lunghezza massima del vettore dei token di inizio/fine e la lunghezza massima del vettore dei cammini[1] (vedremo in seguito che valori sono stati assegnati).

[1]Nel caso in cui non siano effettivamente lunghi l o k vengono ridimensionati tramite del *padding*

Prima però di poter utilizzare questo vettore come input del modello, esso deve essere trasformato in tre vettori separati della seguente forma:

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

dove la constante c rappresenta la lunghezza massima dei vettori di *ast contexts* (di nuovo in seguito vedremo i suoi valori). Definiamo i tre vettori nel seguente modo:

$$\begin{aligned} x_s &= (x_s^{(0)}, x_s^{(1)}, ..., x_s^{(c)}) \\ x_t &= (x_t^{(0)}, x_t^{(1)}, ..., x_t^{(c)}) \\ p &= (p^{(0)}, p^{(1)}, ..., p^{(c)}) \end{aligned}$$

Può succedere, però, che un vettore di *ast contexts* abbia una lunghezza $c' < c$. In questo caso, dovremmo andare ad aggiungere $c - c'$ *ast contexts* di *padding* che saranno rappresentati da specifiche triple di vettori di token. Questi token, nel rispettivo vocabolario, rappresentano token speciali di *padding*.
Fatto ciò dobbiamo indicare al modello quali degli *ast contexts* sono di *padding*. Per far questo introduciamo l'ultimo dei quattro input del modello: la maschera. La maschera sarà un vettore m di lunghezza c definito nel seguente modo:

$$m_i = \begin{cases} 1 & \text{se l'elemento } i\text{-esimo non è padding} \\ 0 & \text{altrimenti} \end{cases}$$

3.1.2 Gestione cambiamenti della forma

Una volta trasformato l'input, avremo tante quadruple della forma:

$$(x_s, p, x_t, m)$$

tale per cui:

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

All'interno del modello, la prima operazione che avviene è quella dell'*embedding* dei tre vettori di token attraverso un *layer* specifico. Il risultato di ciò sono dei vettori della forma:

$$x'_s, x'_t \in \mathbb{N}^{c \times l \times d}, \quad p' \in \mathbb{N}^{c \times k \times d}$$

dove d è la dimensione dell'*embedding* (nota: d può essere diverso per p, x_s e x_t). Nello studio di Code2Vec [2], come era già stato discusso nel Capitolo 2, i vettori d'input hanno una forma leggermente diversa:

$$x_s, x_t \in \mathbb{N}^c, \quad p \in \mathbb{N}^c, \quad m \in \mathbb{N}^c$$

ottenendo successivamente al *layer* di *embedding*:

$$x'_s, x'_t \in \mathbb{N}^{c \times d}, \quad p' \in \mathbb{N}^{c \times d}$$

Per uniformare quindi i valori a quelli usati dalla ricerca, effettueremo un appiattimento dei vettori post-*embedding*, ottenendo:

$$x''_s, x''_t \in \mathbb{N}^{c \times (l \cdot d)}, \quad p'' \in \mathbb{N}^{c \times (k \cdot d)}$$

3.1.3 Classificazione

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

$$0 \leq c_i \leq 1$$

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, cioè:

$$\arg\max_i c_i$$

Nel lavoro svolto, la rete di classificazione prenderà in input il vettore delle *feature* prodotto dal modello di Code2Vec che, in seguito, verrà dato in input ad un *hidden dense layer* culminante in un *layer* di predizione. Quest'ultimo utilizza come funzione di attivazione la funzione *softmax*, andando a produrre il vettore c.
Visto l'output che produce questo modello, prima di poter computare la funzione di *loss*, dovremo trasformare il *label* associato al *code snippet* in una versione *one-hot encoded*.

3.1.4 Regressione

Il modello della regressione ha come scopo quello di predire il numero della riga dell'eventuale errore. La struttura utilizzata è molto semplice: un unico *dense layer* che prende in input il vettore delle *feature* con un singolo output.
Similmente alla classificazione, anche per la regressione dobbiamo processare la riga dell'errore associata al *code snippet*. Per evitare di avere una varianza troppo grande, con a volte numeri di riga molto bassi e a volte molto alti, il valore viene normalizzato da un fattore costante m tale da rendere ogni singolo valore compreso tra 0 e 1.

3.2 Sbilanciamento del dataset

Illustriamo ora il problema principale in cui ci si è imbattuti nel realizzare questo modello: lo sbilanciamento del dataset. Un dataset, in un problema di classificazione, si definisce sbilanciato se le proporzioni del numero di campioni di ogni classe esibiscono grosse differenze. Nel nostro caso possiamo vedere ciò in Tabella 3.1. Come si può notare, la classe dell'assenza di errori rappresenta circa il 92% del dataset, mentre il restante 8% è suddiviso fra le 16 classi possibili di errori. Il problema dello sbilanciamento è molto grave poiché rende difficile sia l'addestramento della rete, sia la sua valutazione. Vediamo un esempio di tutto ciò: supponiamo di creare un modello che predice, per ogni input datogli, sempre l'assenza di errore; col nostro dataset questo modello avrebbe una precisione del 95%. Utilizzando solo questa metrica potrebbe quindi sembrare essere un modello quasi ideale, mentre invece non lo è.

Classe	N. osservazioni	Percentuale
0	41417	92,1%
1	1032	2,3%
2	911	2,0%
3	792	1,8%
4	325	0,7%
5	122	0,3%
6	85	0,2%
7	83	0,2%
8	80	0,2%
9	56	0,1%
10	31	0,06%
11	16	0,03%
12	10	0,02%
13	4	0,008%
14	3	0,006%
15	2	0,004%
16	1	0,002%

Tabella 3.1: Tabella che mostra quanto il dataset sia sbilanciato sia verso la classe dell'assenza di errori sia internamente fra le classi di errori

L'addestramento, invece, è reso difficile dal momento che, con le funzioni di *loss* utilizzate, il modello tenderà naturalmente a diventare come quello descritto sopra. Verranno quindi utilizzate una serie di tecniche nel tentativo di ridurre gli effetti dello sbilanciamento, in particolare vedremo:

- L'utilizzo di una funzione di *loss* pesata.
- L'utilizzo di *oversampling*.
- La riduzione del numero di classi da predire.

Nell'addestramento del modello finale verranno utilizzate sia la seconda sia la terza tecnica.

3.2.1 Funzione di costo pesata

Il meccanismo di funzione di *loss* pesata lavora in base ad un principio molto semplice: assegnare ad ogni classe peso diverso nella computazione della funzione di costo. Così facendo, se si sono assegnati i pesi corretti, si avrà che le classi minoritarie avranno molto più peso rispetto a quelle maggioritarie e quindi, nel caso in cui il modello sbagli a predire una delle classi minoritarie, la perdita sarà maggiore.
In questo lavoro, la funzione di costo pesata è stata implementata attraverso l'utilizzo di una *matrice dei pesi* W della forma:

$$W \in \mathbb{R}^{n \times n}$$

dove n rappresenta il numero di classi. La semantica di questa matrice W è la seguente: il valore $W_{i,j}$ indica il peso di un *sample* di classe i classificato erroneamente come di classe j. Definendo f_i come la frequenza assoluta della classe i-esima, dovremo avere quindi che:

$$W_{i,j} \propto \frac{f_j}{f_i}$$

Esistono diverse tecniche per assegnare questi pesi. In questo caso è stata utilizzata la seguente formula:

$$W_{i,j} = \frac{f_j + \epsilon}{f_i + \epsilon}$$

L'aggiunta di un piccolo valore ϵ è dovuta al fatto che, in rari casi, $f_i = 0$.
È stato deciso di non utilizzare questo sistema poiché i risultati ottenuti non sono stati ottimali. Nei test effettuati, infatti, il modello imparava in tutti i casi a predire sempre la classe di assenza d'errore. Una possibile spiegazione di ciò è che, nonostante le classi minoritarie avessero un grosso peso, la probabilità di trovarle in un singolo *batch* di addestramento era bassa. Ciò implicava uno strano comportamento della funzione di *loss*.

3.2.2 Oversampling

L'*oversampling* è il processo utilizzato per aumentare artificialmente il numero di osservazioni delle classi minoritarie in modo tale da pareggiarle con quelle maggioritarie. L'implementazione dell'*oversampling* può avvenire in svariati modi:

- Ripetizione semplice delle osservazioni delle singole classi minoritarie.
- Creazione di osservazioni completamente nuove tramite metodi complessi. Un esempio di uno degli approcci più popolari è il metodo denominato *smote*, descritto nella ricerca [4], che consiste nel creare dati nei segmenti che congiungono i *k-nearest neighbors* della stessa classe nel *feature space*.

In generale, utilizzando questa tecnica avremo che l'addestramento del modello diventa più difficile poiché è più probabile che finisca in *overfitting*. Creando però dati completamente nuovi, ma teoricamente sensati, cioè come fa *smote*, la probabilità di fare *overfitting* è minore.
Nel caso di questo progetto non è stato possibile utilizzare *smote* poiché l'implementazioni nelle librerie più comuni non funzionavano per la struttura dati usata. Viene, invece, utilizzato il seguente metodo: ripetizione dei dati mischiando però l'ordine degli *ast*

contexts all'interno del vettore. I due vantaggi ottenibile teoricamente tramite questa tecnica sono:

- Diminuire l'*overfitting* riducendo il numero di dati uguali.
- L'eliminazione della semantica associata all'ordine degli *ast contexts* all'interno di un vettore. Infatti, per come vengono estratti, l'ordine non ha alcuna importanza.

3.2.3 Riduzione del numero di classi

L'ultima tecnica che andiamo ad esporre è la riduzione del numero di classi da predire. Questa tecnica trasforma il problema di classificazione a n classi in un problema di classificazione a $n' < n$ classi. Una volta fissato un $n' < n$ verranno determinate le $n' - 1$ classi più frequenti (cioè con il numero di osservazioni più alto) e verranno scelte come le nuove classi. Le restanti classi vengono aggregate in un'unica classe indicante un errore sconosciuto. Avremo, quindi, al variare di n' i seguenti casi:

- Ponendo $n' = 2$ avremo un problema di classificazione binaria, in cui viene classificato la presenza o assenza di errore.
- Ponendo $n' = 6$, come utilizzato in questo progetto, avremo la classificazione degli errori più comuni (*memory leak*, *null dereference*, *dead store* e *variable not initialized*), mentre il restante viene classificato come errore sconosciuto.
- Ponendo n' più vicino al valore di n non avremo grossi cambiamenti.

Possiamo vedere in Tabella 3.2 come al variare di n' cambi la distribuzione delle classi. Utilizzando questa tecnica insieme all'*oversampling* si riduce significativamente l'*overfitting*, poiché lo sbilanciamento del dataset è minore.

(a) Riduzione del numero di classi a $n' = 2$

Classe	N. osservazioni	Percentuale
0	41417	92,1%
1	3553	7,9%

(b) Riduzione del numero di classi a $n' = 6$

Classe	N. osservazioni	Percentuale
0	41417	92,1%
1	1032	2,3%
2	911	2,0%
3	792	1,8%
4	325	0.7%
5	493	1,1%

Tabella 3.2: Comparazione tra riduzione a $n' = 6$ e $n' = 2$ classi

3.3 Addestramento

L'addestramento della rete è stato eseguito numerose volte provando valori per gli iperparametri ogni volta diversi. Questi parametri, nel nostro caso, consistono in:

- Gli iperparametri standard come *learning rate, batch size, epochs, steps per epoch* e *dropout rate* (per il *layers* di *dropout*).
- La dimensione degli *embedding* dei token dei cammini e d'inizio/fine, cioè il valore d introdotto in sottosezione 3.1.2.
- La dimensione del vettore delle *feature* prodotto dal modello code2vec.
- Il numero k di classi da utilizzare.

L'addestramento, inoltre, è stato eseguito utilizzando come ottimizzatore l'algoritmo Adam, descritto più in dettaglio in [8]. Come *loss function* ne vengono utilizzate due, una per la regressione e una per la classificazione, e sono le seguenti:

- Per la classificazione nel caso si utilizzi $n' > 2$ la *categorical crossentropy loss*, mentre se $n' = 2$ si utilizza la *binary crossentropy loss*.
- Per la regressione viene utilizzata la *mean squared error*.

Il dataset, prima di essere utilizzato, viene separato in tre insiemi distinti:

- Il *train set*, che è l'effettivo dataset con cui viene addestrata la rete. Questo rappresenta circa l'80% del dataset intero.
- Il *validation set* con cui, ad ogni epoca, si va a valutare l'addestramento della rete. È fondamentale utilizzarlo poiché le metriche che si ottengo sul *train set* non sono affidabili poiché la rete potrebbe fare *overfitting*. Questo rappresenta circa il 10% del dataset.
- Il *test set* con cui, a fine addestramento, si valuta di nuovo la rete. Come il *validation set*, rappresenta circa il 10% del dataset.

Requisito fondamentale è che questi tre insiemi devono essere distinti, e cioè non avere elementi in comune. Se avessero elementi in comune le metriche ottenute non sarebbero più affidabili.

3.3.1 Overfitting

Un modello si dice che è in *overfitting* quando si adatta troppo ai dati osservati e quindi perde capacità di generalizzazione. In particolare, nell'ambito del *Machine Learning* e *Deep Learning*, succede quando il modello ha risultati molto buoni sul *training set*, ma significativamente peggiori sul *validation* e *test set*.
Possiamo vedere un esempio creando un modello che cerca di approssimare una funzione dati dei suoi punti (cioè un problema di regressione). In Figura 3.2 possiamo vedere un modello in *underfit* (cioè il contrario di *overfit*), un modello ideale che ha approssimato correttamente la funzione e un modello in *overfit*. Se utilizzassimo quest'ultimo modello per predire nuovi valori, ad esempio determinando il valore di $f(x)$ dove x non fa parte dei punti, esso non sarebbe in grado di generarne di corretti, mentre il modello ideale sì.

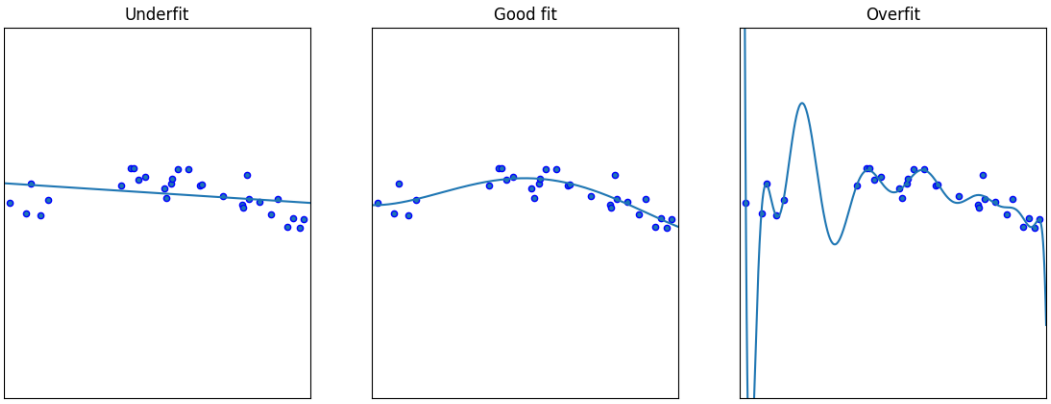

Figura 3.2: Esempio di modello in stato di overfitting, underfitting e ottimale

Questo stato può essere causato da una serie di fattori:

- La complessità del modello, cioè il numero di parametri interni, è troppa alta rispetto al numero di osservazioni nel dataset.
- La fase di addestramento è stata eseguita per troppo tempo.
- Il dataset utilizzato per l'addestramento è troppo piccolo.

Si può rilevare l'*overfitting* guardando l'evoluzione della funzione di costo attraverso l'epoche di addestramento: quando le *loss* calcolate sul *validation* e *train set* divergono, allora è molto probabile essere in *overfitting*. Vediamo un esempio di grafico che ci mostra la divergenza fra le due in Figura 3.3.

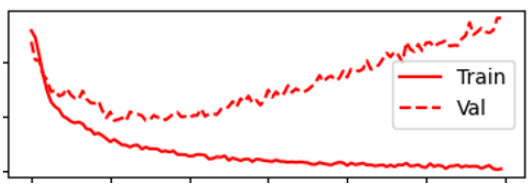

Figura 3.3: Divergenza fra loss nel validation e train set

Il modello utilizzato in questo lavoro soffre altamente di *overfitting* e le cause sono principalmente la dimensione del dataset e la complessità del modello. La prima causa è difficilmente eliminabile visto che andrebbe generato un nuovo dataset, mentre la seconda causa viene affrontata riducendo valori come la dimensione degli *embedding*, del vettore delle *features* e degli input. In particolare, l'ultimo viene fatto attraverso la riduzione dei valori di c, l e k introdotti in sottosezione 3.1.2. Un'altra metodologia utilizzata per ridurlo è l'utilizzo di specifici *layers* di *dropouts*.

3.3.2 Metriche utilizzate

Le metriche sono un meccanismo fondamentale nella valutazione di un modello. Per questo progetto vengono utilizzate principalmente metriche per la classificazione, mentre per la regressione viene utilizzato solo il valore della funzione di costo. In particolare, vengono utilizzate le seguenti metriche:

- La precisione per ogni singola classe i calcolata nel seguente modo:

$$P_i = \frac{TP_i}{TP_i + FP_i}$$

  dove TP_i e FP_i rappresentano rispettivamente il numero di *true positive* e *false positive* per la classe i. La precisione identifica quanti elementi classificati come di classe i sono effettivamente di classe i.

- Il recall per ogni singola classe i calcolato come:

$$R_i = \frac{TP_i}{TP_i + FN_i}$$

  dove FN_i rappresenta il numero di *false negative* per la classe i. Il recall rappresenta quanti degli elementi della classe i sono stati effettivamente individuati.

- L'f1-score per ogni classe i che si calcola nel seguente modo:

$$F1_i = \frac{2 \times P_i \times R_i}{P_i + R_i}$$

e rappresenta la media armonica fra la precisione e il recall. Viene quindi utilizzata per bilanciare recall e precisione.

- La matrice M di confusione in cui ogni elemento $M_{i,j}$ rappresenta il numero di volte nella quale il modello ha predetto la classe j mentre la classe corretta era i. Le predizioni corrette sono quindi rappresentate nella diagonale, cioè quando $i = j$.

Spesso succede che ci sia un *trade off* tra la precisione e il recall. In base al tipo di problema che si sta cercando di risolvere, verrà prediletta una delle due metriche. In questo lavoro si è deciso di prediligere il recall poiché è, a nostro avviso, più importante essere in grado di rilevare tutti i possibili errori.

3.4 Risultati

In questa sezione verranno illustrati i risultati ottenuti dall'addestramento della rete. Tutti i risultati mostrati, come metriche e matrici, sono calcolate sul *test set*. Analizzeremo prima la classificazione a 6 classi. In seguito, vedremo i risultati di quella a 2 classi e infine i risultati della regressione.

3.4.1 Risultati ottenuti con classificazione a 6 classi

La Tabella 3.3 mostra i risultati ottenuti utilizzando un singolo *hidden dense layer* di dimensione 50. Sono stati inoltre utilizzati i seguenti valori: *learning rate* = 0.01, *dropout rate* = 0.3, $d = 128$, $c = 200$, $l = 10$, $k = 20$ e la dimensione del vettore delle *feature* = 64.
Sempre dal risultato dello stesso addestramento possiamo vedere la matrice di confusione nella Figura 3.4. Come si può vedere, è difficile trarre conclusioni significative da essa. Per questo motivo introduciamo la matrice in Figura 3.5, ottenuta normalizzando ogni singola cella dividendo il suo valore per la somma della riga intera (questa è un specie di matrice del *recall*). In quest'ultima matrice sorge in rilievo la diagonale, indicazione delle predizioni corrette del modello.

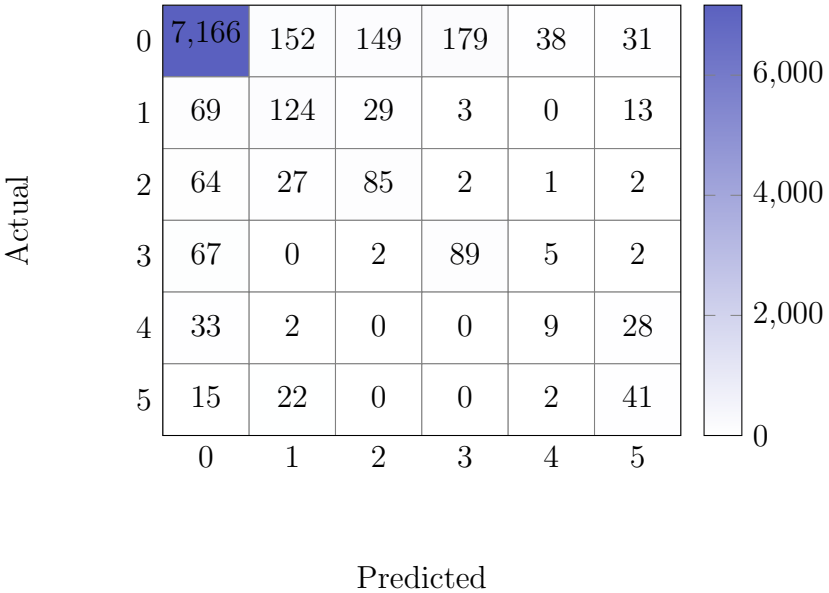

Figura 3.4: Matrice di confusione per $n' = 6$ classi

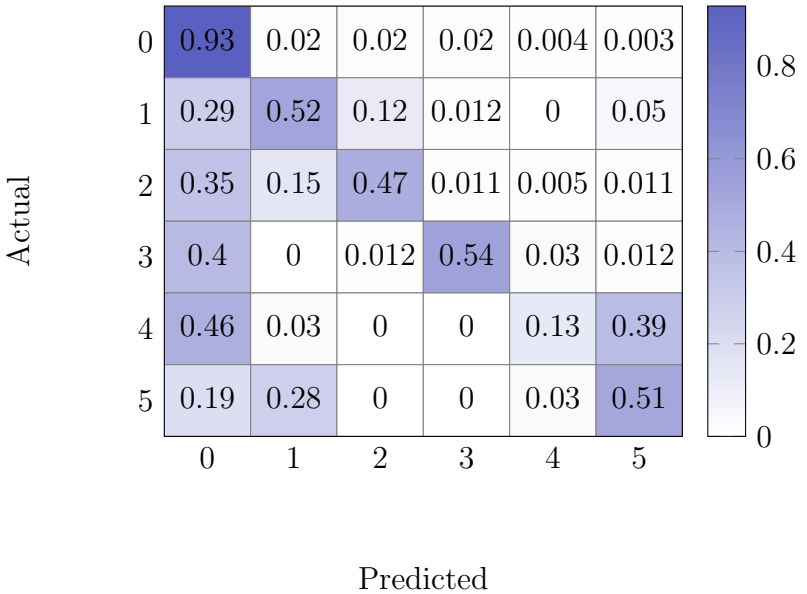

Figura 3.5: Matrice di confusione normalizzata per $n' = 6$ classi

Classe	F1-score	Recall	Precisione
0	0.95	93%	97%
1	0.44	52%	38%
2	0.38	47%	32%
3	0.41	54%	33%
4	0.14	13%	16%
5	0.42	51%	35%
Generale	0.46	52%	42%

Tabella 3.3: Tabella dei risultati con $n' = 6$

Sia dalla matrice sia dalla tabella, possiamo vedere come il modello è in grado di determinare efficacemente l'assenza di errori, mentre si confonde di più sul predire la classe di essi. Se però analizziamo la tabella non considerando la classe dell'assenza di errore, possiamo notare come il modello riesca a distinguere molto bene fra le varie classi di errori (tranne un po' di confusione nella classe 4).

3.4.2 Risultati ottenuti con classificazione a 2 classi

In questa sezione vediamo i risultati ottenuti con una classificazione a 2 classi, cioè una classificazione binaria atta a determinare la presenza o assenza di errore. L'addestramento è stato fatto utilizzando gli stessi iperparametri usati nella sezione sopra. Nella Tabella 3.4 possiamo vedere come il modello ottiene risultati migliori rispetto alla classificazione con più classi. Ciò si riflette anche sulla matrice di confusione in Figura 3.6 e Figura 3.7

Classe	F1-score	Recall	Precisione
0	0.95	96%	95%
1	0.54	51%	57%
Generale	0.75	73%	76%

Tabella 3.4: Tabella dei risultati con $n' = 2$

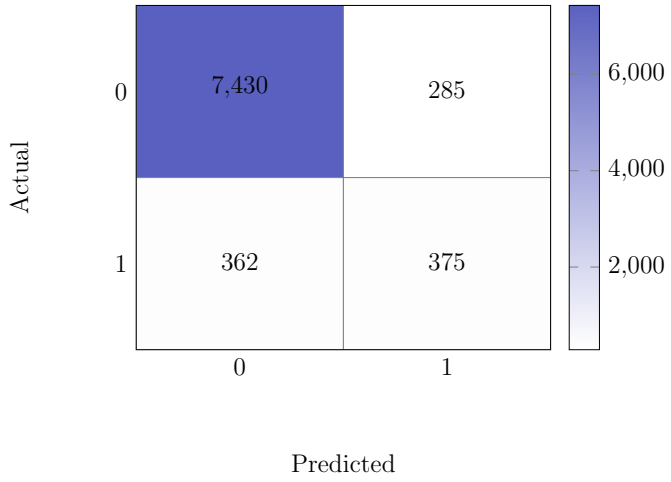

Figura 3.6: Matrice di confusione della classificazione binaria

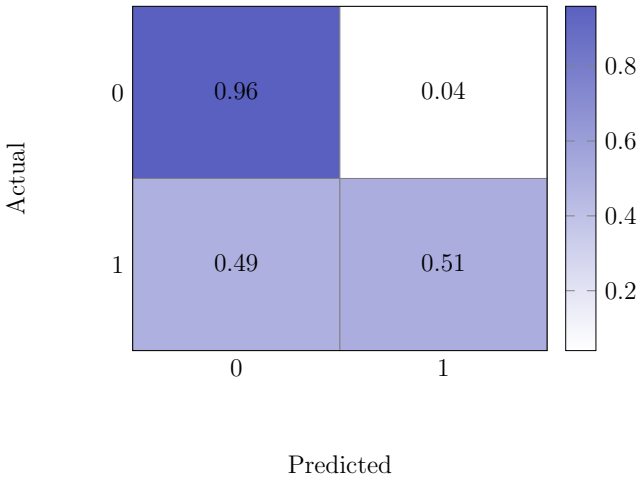

Figura 3.7: Matrice di confusione della classificazione binaria normalizzata

3.4.3 Risultati della regressione

I risultati della regressione sono molto simili, sia addestrando la classe per la classificazione a 6 classi, sia a 2 classi. Come già detto, per valutarla viene utilizzato il *mean squared error*. Data la formula del *mean squared error*:

$$MSE = \sum_{i}^{n} \left(\frac{y_i}{m} - \frac{\hat{y}_i}{m} \right)^2$$

dove m rappresenta il fattore di normalizzazione, utilizzato per trasformare l'intervallo ad $[0, 1]$, otteniamo in seguito che:

$$RMSE = \sqrt{m^2 * MSE} = m * \sqrt{MSE}$$

dove $RMSE$ rappresenta il *root mean squared error*. Questo valore sta ad indicare quanto la predizione del numero di riga è mediamente distante dal valore vero. Dopo l'addestramento, otteniamo un valore di $MSE \approx 0.009$ che, utilizzando $m = 100$, corrisponde ad un $RMSE \approx 9.5$. Come si può vedere, il modello non è in grado di produrre risultati ottimali. Ciò può essere causato dal duplice lavoro che il modello deve effettuare, cioè classificazione e regressione, rendendo difficile bilanciare correttamente i due (in questo lavoro, inoltre, è stato dedicato più tempo nell'ottenimento di una classificazione corretta poiché ritenuta più importante).

3.5 Ulteriore architettura per la classificazione provata

Come abbiamo visto, il modello riesce a determinare con sufficiente correttezza se un *code snippet* presenta o no un errore e riesce a catalogare bene il tipo di errore. Se, però, deve fare queste predizioni tutte insieme (e cioè deve predire o la classe indicante l'assenza di errore o la classe dell'errore), il modello non generalizza altrettanto bene.
Per provare a migliorare i risultati del modello, è stato provato a dividere il meccanismo

di predizioni in due porzioni:

- Determinare la presenza o l'assenza di un errore.
- Determinare la classe dell'errore.

In questa modalità qui, quindi, il modello, oltre ad effettuare le regressione, esegue due classificazioni: una binaria e una a più classi.
I risultati prodotti da questa versione, però, non sono così distanti dal modello effettivamente usato. Questo potrebbe essere determinato da un fattore principale: la difficoltà nell'addestramento. Infatti, in questo caso il modello deve bilanciare tre predizioni diverse: le due elencate prima e la regressione.
Una possibile miglioria sarebbe quella di separare completamente i due modelli: uno predice la presenza o no di errori e l'altro predice solamente l'errore e il numero della riga. Il funzionamento sarebbe poi il seguente: si utilizza inizialmente il primo modello. Poi, nel caso di rilevamento di errori, si utilizza il secondo modello per determinarne il tipo. Non è stata esplorata questa possibilità poiché al di fuori della portata di questo lavoro.

Capitolo 4

Conclusioni

In questo elaborato è stata discussa la creazione di un modello per il rilevamento di errori nel codice di programmi scritti in linguaggii C e C++. La necessità di sviluppare un modello di questo genere, quando nel campo sono ormai diffusi analizzatori sia statici sia dinamici, deriva dal tentativo di ridurre il numero di falsi positivi rilevati da essi. Il lavoro è stato svolto seguendo la falsariga della ricerca [2], nella quale viene sviluppato un modello, sempre in *Deep Learning*, in grado di processare codice in un formato di vettore di *ast contexts*. In particolare, questo progetto si è sviluppato in due fasi distinte: la generazione del dataset e la creazione del modello.
Per la generazione del dataset è stata utilizzata come base una collezione di progetti, risultato del lavoro descritto nella ricerca [6]. Per ogni progetto sono stati generati dei report di analisi statica utilizzati per estrarre le coppie $\langle$*code snippet, errore*$\rangle$. I *code snippet* sono poi stati processati al fine di creare dei vettori di *ast contexts*.
La seconda parte del progetto consisteva nello sviluppo del modello predittivo, il quale obbiettivo è duplice: predire la classe di errore e predire il numero della riga dell'eventuale errore. Abbiamo poi visto il problema principale riscontrato nell'addestramento, lo sbilanciamento del dataset, ed una serie di metodi per ovviarlo. In seguito è stato anche esplorato il problema del *overfitting*. Come, però, abbiamo visto nella sezione 3.4, i risultati ottenuti non sono entusiasmanti. Questo è dovuto ad una combinazione di più fattori:

- La dimensione ridotta del dataset.
- La difficoltà nell'addestramento del modello che spesso finisce in stati di *overfitting*.
- La complessità intrinseca del problema. Infatti, anche per un programmatore esperto potrebbe essere difficile identificare correttamente certi errori.

4.1 Miglioramenti possibili e sviluppi futuri

Dai risultati ottenuti possiamo quindi concludere che il modello, come sviluppato in questo progetto, non è sufficientemente capace di predire gli errori. Possibili miglioramenti riguarderebbero:

- La dimensione del dataset che, nonostante non sia di piccole dimensioni, potrebbe non essere adatto per una generalizzazione corretta del modello.
- La struttura del modello stesso. Infatti, come abbiamo visto, il modello è difficilmente addestrabile ed eccessivamente complesso.

Soluzione alternativa per la struttura del modello potrebbe essere quella descritta nella ricerca [3], nella quale viene generata una rappresentazione del codice attraverso l'uso di un *Tree-Based Convolutional Neural Network*. Questo modello viene messo a disposizione già addestrato. Il *training* in questo caso potrebbe rivelarsi più semplice, poiché deve essere imparata solamente la classificazione e la regressione, ma non la porzione di *encoding* del codice.

Bibliografia

[1] Elena N Akimova et al. “A survey on software defect prediction using deep learning”. In: *Mathematics* 9.11 (2021), p. 1180.

[2] Uri Alon et al. “Code2Vec: Learning Distributed Representations of Code”. In: *Proc. ACM Program. Lang.* 3.POPL (gen. 2019), 40:1–40:29. ISSN: 2475-1421. DOI: 10.1145/3290353. URL: http://doi.acm.org/10.1145/3290353.

[3] Nghi DQ Bui, Yijun Yu e Lingxiao Jiang. “Infercode: Self-supervised learning of code representations by predicting subtrees”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1186–1197.

[4] Nitesh V Chawla et al. “SMOTE: synthetic minority over-sampling technique”. In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[5] Lloyd D Fosdick e Leon J Osterweil. “Data flow analysis in software reliability”. In: *ACM Computing Surveys (CSUR)* 8.3 (1976), pp. 305–330.

[6] Ben Gelman et al. “Source code analysis dataset”. In: *Data in brief* 27 (2019), p. 104712.

[7] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[8] Diederik P Kingma e Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[9] Vladimir Kovalenko et al. “PathMiner: a library for mining of path-based representations of code”. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press. 2019, pp. 13–17.

[10] Michail Papamichail, Themistoklis Diamantopoulos e 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)*. IEEE. 2016, pp. 100–107.

[11] Tushar Sharma et al. “A survey on machine learning techniques for source code analysis”. In: *arXiv preprint arXiv:2110.09610* (2021).