UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Confronto tra classificazione metric-based e NLP-based per l'individuazione di Code Smell

RELATORE

Prof. Andrea De Lucia

Dott. Manuel De Stefano

Università degli Studi di Salerno

CANDIDATO

Dario De Maio

Matricola: 0512109837

Anno Accademico 2022-2023

Questa tesi è stata realizzata nel

Fatti non foste a viver come bruti, ma per seguir virtute e canoscenza

Abstract

I code smells rappresentano scelte implementative mediocri fatte dagli sviluppatori nel codice sorgente, compromettendo la manutenibilità. Questo attributo di qualità è fondamentale per garantire l'affidabilità ed efficienza a lungo termine dei prodotti software. La rimozione dei code smell riveste un ruolo cruciale poiché contribuisce alla riduzione del "debito tecnico", ottimizzando le future attività di manutenzione e contenendo i costi. Tuttavia, individuare i code smell risulta particolarmente complesso a causa di molteplici variabili, tra cui la soggettività degli sviluppatori. Per affrontare tali sfide, sono state introdotte varie tecniche, ciascuna con approcci distinti, per l'identificazione automatica dei code smell. Molti approcci per l'identificazione dei code smells si basano su euristiche, che combinano una serie di metriche per creare regole che determinano la presenza o l'assenza di un code smell. Sebbene queste euristiche sembrino essere abbastanza accurate nelle classificazioni, esse presentano alcune limitazioni, in particolare: la soggettività degli sviluppatori, la mancanza di accordo tra diversi rilevatori, e la difficoltà nel trovare soglie universali, poiché le regole sono approssimative. Si è provato in letteratura a superare queste limitazioni tramite l'uso di tecniche basate sull'apprendimento automatico, come il Machine Learning. Tuttavia, nonostante i diversi sforzi le tecniche attuali non sono ancora in grado di dare risultati ampiamente affidabili. In questo progetto, l'obiettivo è utilizzare modelli di Machine Learning basati sull'elaborazione del linguaggio naturale (NLP) come tentativo di superare le difficoltà che gli approcci tradizionali hanno dimostrato di avere. Questi modelli, come ad esempio "CodeBert", consentono di analizzare intere porzioni di codice e comprenderlo. Ciò permette di analizzare e comprendere il contesto in modo più efficiente, consentendo l'identificazione di code smells che potrebbero essere difficili da riconoscere con modelli di Machine Learning che si basano esclusivamente su metriche. In questo progetto si mettono a confronto entrambe le tecniche. Dopo aver addestrato entrambi i modelli, il primo basato su approcci tradizionali e il secondo basato su NLP, si procede con l'analisi delle loro performance utilizzando tecniche statistiche per valutare la correttezza degli approcci e la loro applicabilità in contesti reali, nonché stimare quale dei due

porta risultati più promettenti. In sintesi, l'obiettivo principale di questo progetto è migliorare l'identificazione dei code smells utilizzando modelli di Machine Learning basati sull'elaborazione del linguaggio naturale. Ciò consentirà di superare le limitazioni degli approcci tradizionali basati su euristiche, migliorando la manutenibilità del codice e garantendo prodotti software più affidabili ed efficienti nel tempo.

Indice

Elenco delle Figure **iii**

Elenco delle Tabelle **iv**

1 Introduzione **1**

1.1 Contesto applicativo 1

1.2 Motivazione e obiettivi 2

1.3 Risultati ottenuti 3

2 Background e stato dell'arte **4**

2.1 Introduzione 4

2.2 Code Smells e tecniche per riconoscerli 7

2.2.1 Code Smells 7

2.2.2 Approcci per l'identificazione di Code Smells 9

2.2.3 Rilevamento di Code Smells con euristiche 10

2.2.4 Tecniche di Machine Learning per l'identificazione di Code Smells 11

2.2.5 Tecniche basate su NLP 13

3 Strategia di ricerca **17**

3.1 Overview dell'approccio di ricerca 17

3.2 Contesto applicativo 18
3.3 Setup 21
3.3.1 Panoramica generale 21
3.3.2 Creazione del dataset finale 22
3.3.3 Preparazione dei dati 25
3.3.4 Scelta dei modelli di Machine Learning tradizionale e modelli di NLP 31
3.3.5 Addestramento dei modelli 34
3.3.6 Metriche di valutazione 35

4 Analisi dei risultati **39**
4.1 Overview dell'analisi dei risultati 39
4.2 Analisi dei risultati 40
4.2.1 Risultati per Class Data Should Be Private 45
4.2.2 Risultati per Large Class 47
4.2.3 Risultati per Complex Class 47
4.2.4 Risultati per Refused Bequest 48
4.2.5 Risultati per Lazy Class 49
4.2.6 Risultati per Spaghetti Code 49
4.2.7 Minacce alla validità 51

5 Conclusioni e sviluppi futuri **52**
5.1 Conclusioni 52
5.2 Sviluppi futuri 53

Bibliografia **55**

Elenco delle figure

3.1 Class Data Should Be Private 28
3.2 Long Class 29
3.3 Lazy Class 29
3.4 Refused Bequest 30
3.5 Spaghetti Code 30
3.6 Complex Class 31

4.1 MCC: Class Data Should Be Private 40
4.2 MCC: Large Class 40
4.3 MCC: Lazy Class 40
4.4 MCC: Refused Bequest 40
4.5 MCC: Spaghetti Code 41
4.6 MCC: Complex Class 41
4.7 F1: Class Data Should Be Private 41
4.8 F1: Large Class 41
4.9 F1: Lazy Class 41
4.10 F1: Refused Bequest 41
4.11 F1: Spaghetti Code 42
4.12 F1: Complex Class 42

Elenco delle tabelle

3.1 Progetti considerati nel contesto dello studio 19
3.2 Code Smell considerati nel contesto dello studio 20

4.1 p-value ottenuti dal test di Friedman 43
4.2 p-value ottenuti dal test di Friedman 44
4.3 p-value relativi a CDSBP dopo aver effettuato la correzione di Bonferroni su McNemar 46
4.4 p-value relativi a LC dopo aver effettuato la correzione di Bonferroni su McNemar 47
4.5 p-value relativi a CC dopo aver effettuato la correzione di Bonferroni su McNemar 48
4.6 p-value relativi a RB dopo aver effettuato la correzione di Bonferroni su McNemar 49
4.7 p-value relativi a SC dopo aver effettuato la correzione di Bonferroni su McNemar 50

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

Lo sviluppo di un software corretto è un processo complesso e richiede una solida comprensione delle fasi di progettazione che precedono l'implementazione. Tuttavia, una volta completate queste fasi, spesso si sottovaluta l'importanza della manutenzione del software. La manutenzione è un processo che può essere molto più costoso della fase di sviluppo e può presentare sfide difficili da gestire.

Le difficoltà nella fase di manutenzione spesso sono correlate alla mancanza di controllo durante le fasi di analisi e progettazione del software. Se questi aspetti non sono stati adeguatamente considerati, possono emergere problemi e complicazioni nel processo di manutenzione.

Tra i fattori che possono contribuire alle difficoltà di manutenzione, ci sono anche quelli tecnici. Ad esempio, la comprensione del codice scritto da altri può essere difficile, specialmente se non è stato ben documentato o se è stato scritto senza seguire linee guida standard. Inoltre, il software potrebbe non essere stato progettato tenendo conto delle modifiche future, il che può rendere complessa l'introduzione di nuove funzionalità o il risolvere errori.

Le mancanze durante le varie fasi possono portare all'adozione di scelte implementative mediocri nel codice sorgente. Questi problemi, noti come "Code Smells" in letteratura, possono compromettere la qualità e la manutenibilità a lungo termine del software [1].

Affrontare i Code Smells può essere particolarmente impegnativo per i programmatori, specialmente quando le scadenze sono rigorose e c'è poca disponibilità di tempo. In queste situazioni, è possibile che siano fatte scelte implementative mediocri, il che può peggiorare la situazione nel lungo periodo.

Riconoscere e prevenire l'introduzione di Code Smells è un passo essenziale per assicurare la solidità del software e facilitare la sua manutenzione nel tempo. Evitare scelte implementative discutibili può aiutare a ridurre il rischio di bug, migliorare le prestazioni e rendere il codice più leggibile e comprensibile.

1.2 Motivazione e obiettivi

Come accennato in precedenza, l'incessante pressione delle scadenze sempre più stringenti spinge gli sviluppatori a sacrificare i principi e le linee guida della buona programmazione al fine di implementare rapidamente le richieste di cambiamento. Tuttavia, queste pratiche portano involontariamente all'introduzione di "code smell" all'interno del codice, ovvero segnali di possibili problemi di design e manutenibilità.

Riconoscere e trattare i code smell rappresenta un'impresa complessa, resa tale da molteplici variabili, tra cui la natura soggettiva delle preferenze degli sviluppatori. Questa soggettività può portare a sfide considerevoli nell'individuazione dei problemi nel codice sorgente.

Per tale motivo, sono state introdotte una varietà di tecniche, ciascuna con l'obiettivo di rilevare e identificare questi code smell. Tra le molteplici strategie presenti in letteratura, emergono principalmente due categorie: l'identificazione basata su euristiche e l'approccio basato su Machine Learning. La prima di queste tecniche si fonda su regole euristiche predeterminate, mentre la seconda sfrutta algoritmi di Machine Learning per l'analisi del codice.

Nel contesto delle euristiche, l'identificazione dei code smell avviene attraverso un insieme di criteri definiti manualmente, che evidenziano situazioni potenzialmen-

te problematiche nel codice. Questo approccio richiede una conoscenza approfondita delle best practice di sviluppo e delle tipiche strutture di codice soggette a problematiche. D'altro canto, l'approccio basato su Machine Learning sfrutta modelli algoritmici addestrati su dati per riconoscere automaticamente i pattern associati ai code smell.

Il principale obiettivo di questa tesi è esplorare l'utilizzo di nuove tecniche e tecnologie per l'identificazione dei code smell e successivamente valutare se queste si dimostrano effettivamente migliori rispetto alle tecnologie già presenti in letteratura o comunemente utilizzate.

In particolare, il fulcro di questo studio è rappresentato dall'applicazione di modelli di NLP. La ricerca si concentra sull'impiego di un modello di NLP in grado di identificare i code smell all'interno del codice sorgente, al fine di verificare se tale approccio risulti più efficace rispetto ai modelli tradizionali di machine learning che sono stati oggetto di ricerca in passato.

Oltre agli aspetti tecnici, questa tesi intende fornire un contributo alla comunità di ricerca e agli sviluppatori di software, fornendo una panoramica chiara e dettagliata delle potenziali applicazioni delle tecniche di NLP nell'ambito dell'ingegneria del software.

1.3 Risultati ottenuti

L'esperimento è stato condotto con grande attenzione e rigore scientifico, seguendo il piano prestabilito. Lo scopo principale era valutare l'efficacia del modello di NLP noto come CodeBERT nel rilevare i code smell, confrontandolo con approcci più tradizionali basati su machine learning, come RandomForest che riceve metriche tradizionali come input e un altro modello di RandomForest dove gli input consistevano in vettorizzazioni di codice sorgente.

Il confronto tra CodeBERT e i due modelli di RandomForest ha consentito di ottenere una panoramica completa delle prestazioni dei due approcci. Questo confronto è stato particolarmente rilevante, poiché ha permesso di valutare in modo chiaro i vantaggi e le limitazioni di un modello di NLP rispetto alle tecniche tradizionali di machine learning nella rilevazione dei code smell.

CAPITOLO 2

Background e stato dell'arte

2.1 Introduzione

La manutenzione dei software e l'evoluzione di essi rappresentano attività complesse e critiche nell'ambito dello sviluppo software. Durante il ciclo di vita di un'applicazione, è inevitabile che si rendano necessarie modifiche al codice sorgente per soddisfare nuovi requisiti, correggere errori, o migliorare le prestazioni. Tuttavia, le pressioni del tempo e le scadenze stringenti spesso costringono gli sviluppatori a effettuare cambiamenti rapidi, portando talvolta all'adozione di scelte implementative mediocri. Queste scelte, notoriamente chiamate "Code Smells" nella letteratura, rappresentano segnali di potenziali problemi nel design o nell'organizzazione del codice.

I code smells sono indicativi di componenti del codice sorgente che possono essere raffinate e ottimizzate. Identificare e correggere i code smells in modo tempestivo è fondamentale per mantenere la manutenibilità e la qualità del software nel lungo periodo. Nel corso degli anni, sono state proposte diverse categorie di code smells, come il "God Class" (classe di dio), il "Long Method" (metodo troppo lungo), il "Complex Class" (classe complessa) e molti altri [2][3]. Ognuna di queste categorie indica una potenziale problematica nel codice e richiede una specifica attenzione da

parte degli sviluppatori.

Inizialmente, l'identificazione dei code smells era un'attività soggettiva e basata sull'esperienza degli sviluppatori [4]. Tuttavia, questa approccio ha dimostrato di essere influenzato da diversi fattori, come lo stile di programmazione personale e la familiarità con determinate tecnologie [3] [4]. Per superare queste limitazioni, sono state sviluppate tecniche euristiche per l'identificazione automatica dei code smells. Questi approcci si basano su un insieme di metriche software, come la complessità ciclomatica, la lunghezza del metodo e il numero di dipendenze, e applicano soglie predefinite per determinare la presenza o l'assenza di un code smell [3].

Nonostante i vantaggi delle tecniche euristiche, come la semplicità di implementazione e l'applicabilità generale, presentano alcuni svantaggi significativi. Innanzitutto, la soggettività degli sviluppatori può influenzare l'interpretazione dei code smells e portare a conclusioni diverse tra i membri del team [5]. Inoltre, le soglie utilizzate per identificare i code smells sono spesso dipendenti dal contesto e potrebbero non funzionare in modo efficace in tutti gli scenari [5] [6].

Negli ultimi anni, la ricerca nel campo dell'identificazione dei code smells si è evoluta, esplorando l'uso di tecniche basate su Machine Learning per automatizzare il processo di rilevamento dei code smells. Mentre inizialmente si pensava che queste tecniche potessero superare le limitazioni degli approcci basati su euristiche, gli studi hanno dimostrato che l'applicazione di algoritmi di Machine Learning non è stata una soluzione esaustiva [3] [2]. Utilizzando algoritmi di Machine Learning, è possibile addestrare modelli che apprendono da grandi quantità di dati di codice sorgente annotato con la presenza o l'assenza di code smells. Questi modelli possono generalizzare le relazioni tra le caratteristiche del codice e la presenza dei code smells, consentendo l'identificazione automatica e precisa di tali problemi nel software.

L'utilizzo di tecniche di Machine Learning per l'identificazione dei code smells può teoricamente offrire numerosi vantaggi nel processo di rilevamento di questi problemi nel codice sorgente. Innanzitutto, l'approccio basato su Machine Learning riduce l'influenza delle interpretazioni personali degli sviluppatori nell'individuazione dei code smells. I modelli di Machine Learning sono addestrati su dati oggettivi, fornendo una valutazione basata su caratteristiche del codice e pattern riconosciuti, eliminando così la variabilità introdotta dall'interpretazione soggettiva dei singoli

sviluppatori [5]. Tuttavia, è importante notare che l'utilizzo di tecniche di Machine Learning per l'identificazione dei code smells presenta anche alcune sfide e limitazioni.

Così come i metodi basati su euristiche, l'approccio di Machine Learning si basa su un insieme di variabili indipendenti che rappresentano le caratteristiche rilevanti del codice, come la complessità, la coesione e la coerenza. Queste variabili vengono utilizzate per addestrare i modelli di Machine Learning al fine di apprendere eventuali relazioni complesse tra le caratteristiche del codice e la presenza dei code smells. L'obiettivo è ottenere un'analisi più approfondita e accurata rispetto alle semplici euristiche basate su soglie fisse.

Tuttavia, ci sono anche alcuni svantaggi da considerare nell'utilizzo di tecniche di Machine Learning per l'identificazione dei code smells. Innanzitutto, la scalabilità può essere un problema, poiché l'addestramento di un modello di Machine Learning richiede una grande quantità di dati annotati [3]. La raccolta e l'annotazione di un tale dataset possono richiedere tempo e risorse significative . Inoltre, l'addestramento di modelli complessi può richiedere una notevole potenza di calcolo [7] [8].

Un altro aspetto critico è la dipendenza dai dati di addestramento. La qualità dei risultati del Machine Learning dipende dalla qualità dei dati utilizzati per l'addestramento [2]. Se i dati non sono rappresentativi del problema reale o contengono errori o bias, il modello potrebbe produrre risultati inaccurati o poco affidabili. Pertanto, è fondamentale assicurarsi che il dataset di addestramento sia completo rappresentativo del dominio di applicazione specifico.

Infine, l'elevato sbilanciamento dei dataset può rappresentare una sfida nell'identificazione dei code smells tramite Machine Learning. Poiché alcuni code smells sono meno frequenti di altri nel codice sorgente, i modelli di Machine Learning potrebbero avere difficoltà a riconoscere correttamente le istanze dei code smells minori, poiché l'informazione relativa a tali istanze può essere scarsa nel dataset di addestramento.

Nonostante questi svantaggi, le tecniche basate su Machine Learning per l'identificazione dei code smells stanno guadagnando sempre più attenzione e sono oggetto di ricerca continua. Nelle sezioni successive, presenteremo le definizioni dei code smells studiati in letteratura e gli approcci proposti per la loro identificazione, soffermandosi su quelli che hanno avuto maggiore rilevanza sul lavoro di questa tesi.

2.2 Code Smells e tecniche per riconoscerli

In questa sezione, verranno approfondite due tecniche analizzate in letteratura per l'identificazione dei Code Smell, oltre alle varie classi di Code Smell su cui è stato concentrato il lavoro di tesi [3]. Prima di introdurre le tecniche per il rilevamento dei Code Smell analizzate in letteratura, è importante comprendere quali sono i Code Smell presi in considerazione e sui quali è stata focalizzata maggiormente l'attenzione nella ricerca.

2.2.1 Code Smells

Di seguito vengono riportate alcune delle principali classi di Code Smell:

1. Large Class o Blob: Una "Large class" è una classe che contiene un numero eccessivo di metodi e proprietà, superando di solito una soglia di complessità. Questo code smell indica una progettazione sbagliata, in quanto la presenza di una classe troppo lunga rappresenta una mancata divisione delle responsabilità, e ciò comporta difficile comprensione e manutenzione del codice. La presenza di un gran numero di metodi e proprietà può rendere il codice confuso e difficile da gestire. Per identificare una classe lunga, si può fare riferimento al numero di metodi e proprietà all'interno della classe, cercando di capire se possono essere raggruppati in classi più piccole e specializzate. Dividere una classe lunga in classi più piccole e coerenti aiuta a migliorare la modularità e la comprensibilità del codice, facilitando la sua manutenzione e riducendo la possibilità di errori.

2. Lazy class: Una "Lazy class" è una classe che non svolge alcun compito utile nel contesto del sistema software. Questo code smell si verifica quando una classe è stata creata per un certo scopo, ma poi il suo compito è stato eliminato o è diventato obsoleto. Questo può accadere a seguito di cambiamenti nella progettazione o nell'architettura del sistema. Identificare una classe pigra richiede un'analisi del suo utilizzo all'interno del codice. Se la classe non viene effettivamente utilizzata per svolgere alcuna funzione nel sistema, è possibile considerarla come un code smell e prendere in considerazione l'eliminazione o la riutilizzazione della classe in modo più significativo. Rimuovere le classi

pigre contribuisce a semplificare il codice e a migliorare la manutenibilità e la chiarezza del sistema.

3. Class data should be private: Questo code smell si verifica quando le proprietà di una classe sono dichiarate pubbliche anziché private. Quando le proprietà di una classe sono pubbliche, possono essere modificate da qualsiasi parte del codice, il che può portare a errori difficili da individuare e a una maggiore dipendenza dalle implementazioni interne della classe stessa. Per risolvere questo problema, è necessario modificare la visibilità delle proprietà, rendendole private, e fornire metodi pubblici, noti come getter e setter, per accedere e modificarle in modo controllato. Questo principio di incapsulamento dei dati aiuta a mantenere un controllo rigoroso sullo stato interno della classe e a evitare effetti collaterali indesiderati. La pratica di rendere private le proprietà di classe e fornire metodi pubblici per accedervi viene spesso denominata principio di "information hiding" o "encapsulation".

4. Complex class: Una "Complex class" è una classe che presenta metodi molto complessi. La metrica utilizzata per individuare questo smell è la complessità ciclomatica media pesata, la quale viene utilizzata per valutare la complessità del flusso di controllo di un metodo o di una classe. Una classe complessa può rendere il codice difficile da comprendere e da modificare, poiché ogni modifica potrebbe influire su diversi percorsi di esecuzione. L'alto numero di percorsi di esecuzione può aumentare la probabilità di bug e rendere la classe difficile da testare. Un approccio per affrontare una classe complessa è quello di suddividerla in classi più piccole e specializzate, seguendo il principio di singola responsabilità. Questo aiuta a migliorare la comprensione e la manutenibilità del codice, riducendo la dipendenza da percorsi di esecuzione complessi.

5. Refused Bequest: Il code smell "Refused Bequest" si verifica quando una sottoclasse eredita una classe padre, ma rifiuta alcuni dei suoi metodi o proprietà modificandone la loro implementazione o semplicemente ignorandoli. Questo significa che la sottoclasse decide di non utilizzare o di non fornire l'implementazione dei metodi o delle proprietà ereditati dalla classe padre. Questo

problema può indicare un design errato o una mancanza di corretta pianificazione nell'ereditarietà delle classi. Quando una sottoclasse rifiuta l'ereditarietà di metodi o proprietà dalla classe padre, può significare che la sottoclasse ha una responsabilità o un comportamento differente rispetto alla classe padre, o che alcuni metodi o proprietà ereditati non sono pertinenti per la sottoclasse. Per risolvere questo problema, è necessario rivedere attentamente la gerarchia delle classi e valutare se la sottoclasse dovrebbe davvero ereditare tutti i metodi e le proprietà della classe padre. Potrebbe essere necessario ridefinire o eliminare l'ereditarietà in base alle specifiche esigenze della sottoclasse, creando una gerarchia più coerente e ben progettata. Un'alternativa potrebbe essere l'utilizzo di composizione o di altri meccanismi di astrazione per separare le responsabilità tra le classi.

6. Spaghetti Code: Lo spaghetti code è spesso il risultato di una cattiva progettazione del software o della mancanza di attenzione ai principi di buona progettazione. Può essere causato da una serie di fattori, tra cui l'abuso di funzioni o metodi, l'uso di strutture di controllo complesse e nidificate, l'uso eccessivo di "goto" o di variabili globali, e così via. Il risultato è un codice che è difficile da leggere e da mantenere, che può portare a errori e a un rallentamento dello sviluppo del software. Per risolvere il problema dello spaghetti code, è necessario rivedere il codice e cercare di semplificarlo, separandolo in parti più piccole e più gestibili, eliminando i blocchi di codice ridondanti, utilizzando nomi di variabili e funzioni più descrittivi e usando una struttura ben organizzata e facile da comprendere.

2.2.2 Approcci per l'identificazione di Code Smells

Successivamente, esploreremo le principali tecniche proposte in letteratura per il rilevamento dei Code Smells all'interno del codice sorgente [3].

2.2.3 Rilevamento di Code Smells con euristiche

Gli approcci euristici si concentrano sull'identificazione dei Code Smells mediante l'uso di regole di rilevamento basate su metriche software. Il processo principale di rilevamento si sviluppa in due fasi. Nella prima fase, l'attenzione è posta sull'individuazione di caratteristiche di qualità che definiscono un Code Smell.Queste caratteristiche vengono rappresentate come metriche. Nella seconda fase, le metriche individuate vengono combinate per definire delle regole che consentono l'identificazione dei Code Smells. Le tecniche di rilevamento possono differire in base alle metriche utilizzate, che dipendono dal tipo di Code Smell considerato e da come queste metriche vengono combinate tra loro.

Inizialmente, molti degli approcci utilizzavano la combinazione di operatori logici AND/OR per definire le regole di rilevamento (ad esempio, if $\texttt{LOC} \geq x$). Un esempio di utilizzo di operatori logici è rappresentato da DECOR, uno strumento che si focalizza sull'estrazione di metriche del codice e sul confronto di tali metriche con soglie predefinite per determinare la presenza di Code Smells. DECOR descrive le caratteristiche di una classe affetta da un Code Smell attraverso un insieme di regole, chiamate "rule card", che delineano le caratteristiche intrinseche di una classe influenzata da un odore.

Un'altra tool basato su euristiche è JDeodorant [3], il quale utilizza algoritmi di clustering supervisionato per raggruppare insieme i code smell simili in base alle metriche del codice. Gli algoritmi di clustering supervisionato prendono in considerazione le caratteristiche comuni tra i code smell e li organizzano in gruppi coerenti. Questo aiuta gli sviluppatori a comprendere meglio le relazioni tra i diversi code smell e a identificare le aree critiche in cui intervenire per migliorare la qualità del codice. Inoltre, questo tool utilizza soglie per tagliare i dendrogrammi risultanti. Le soglie consentono di definire un livello di somiglianza o dissomiglianza oltre il quale i code smell vengono separati in gruppi distinti. L'uso di soglie aiuta a definire i confini tra i cluster di code smell e a rendere più significativa l'organizzazione gerarchica dei dendrogrammi.

Nonostante le buone performance, alcune ricerche [9] hanno evidenziato alcune limitazioni di questi approcci euristici:

1. Soggettività degli sviluppatori: la valutazione e l'interpretazione dei Code Smells possono variare tra diversi sviluppatori, portando a risultati soggettivi.

2. Scarso accordo tra i diversi rivelatori: diversi strumenti di rilevamento di Code Smells possono produrre risultati divergenti per lo stesso codice, creando confusione e incertezza.

3. Difficoltà nel trovare buone soglie: stabilire le soglie corrette per le metriche del codice può risultare complesso e soggetto a dibattito, in quanto i Code Smells possono variare da progetto a progetto.

2.2.4 Tecniche di Machine Learning per l'identificazione di Code Smells

L'approccio di identificazione dei code smell con tecniche di machine learning offre numerosi vantaggi rispetto ai metodi tradizionali [3]. In particolare, sfruttando algoritmi di apprendimento automatico, è possibile analizzare grandi quantità di dati di codice in modo efficiente e automatico, individuando pattern e tendenze che potrebbero non essere evidenti a una valutazione umana. Questo processo di identificazione dei code smell può fornire agli sviluppatori un'analisi più approfondita del codice sorgente e una maggiore consapevolezza dei potenziali problemi e delle aree di miglioramento.

Il primo passo nella tecnica di machine learning per l'identificazione dei code smell consiste nella raccolta dei dati pertinenti. Questi dati possono includere il codice sorgente, le metriche di progettazione (come la complessità ciclomatica, il numero di linee di codice, la coesione delle classi, ecc.), i dati di versione e altre informazioni rilevanti.

Dopo la raccolta dei dati, viene eseguito il preprocessamento per pulire, normalizzare e trasformare i dati in un formato adatto all'algoritmo di machine learning selezionato. Questo può includere operazioni come la rimozione di dati mancanti o rumorosi, la standardizzazione delle scale delle variabili e l'ingegnerizzazione delle caratteristiche per estrarre informazioni rilevanti.

Successivamente, viene selezionato un modello di machine learning adatto all'obiettivo del rilevamento dei code smell. Generalmente, vengono utilizzati algoritmi di classificazione, come ad esempio Random Forest o Naive Bayes. La scelta del modello dipende dalle caratteristiche dei dati e dal tipo di task che si desidera svolgere. È importante considerare la natura dei code smell da individuare e le specifiche del dataset utilizzato per l'addestramento.

Una volta selezionato il modello, si procede con l'addestramento utilizzando un set di dati etichettati e validati per assicurarne la veridicità. Nel caso di dati etichettati manualmente, gli sviluppatori hanno identificato i code smell presenti nel codice e li hanno contrassegnati come tali.

Durante il processo di addestramento, il modello impara a riconoscere i pattern di code smell associando le caratteristiche del codice alle etichette corrispondenti. L'obiettivo è ottenere un modello in grado di generalizzare i pattern appresi su nuovi dati e identificare i code smell in modo accurato.

Una volta addestrato il modello, viene testato su un set di dati separato, noto come "test set", per valutare le sue prestazioni. È importante misurare metriche come l'accuracy, la precision, la recall, la F1-score e la MCC per valutare l'efficacia del modello nel rilevamento dei code smell. Queste metriche forniscono un'indicazione di quanto il modello sia in grado di identificare correttamente i code smell rispetto ai dati di test.

Tuttavia, è importante considerare alcune sfide associate all'uso delle tecniche di machine learning per l'identificazione dei code smell. Ad esempio, la soggettività degli sviluppatori [4] può influenzare la corretta etichettatura dei code smell durante il processo di raccolta dei dati etichettati manualmente. Inoltre, può esserci una mancanza di accordo tra diversi rivelatori basati su machine learning, il che significa che possono essere ottenuti risultati diversi a seconda del modello o dell'approccio utilizzato.

A causa di queste sfide, le tecniche di machine learning non sono migliori rispetto alle tecniche basate su euristiche, e sembrano essere ancora lontane dalle prestazioni ottenute dal loro equivalente euristico [3].

2.2.5 Tecniche basate su NLP

L'utilizzo di tecniche di NLP per l'identificazione dei code smells offre un potenziale significativo per migliorare la qualità del software e semplificare il processo di sviluppo e manutenzione. L'automazione fornita dall'elaborazione del linguaggio naturale può accelerare l'analisi del codice e migliorare la precisione nell'individuazione dei code smells.

Nel contesto di questa ricerca, è stata dedicata particolare attenzione alle tecniche di elaborazione del linguaggio naturale (NLP), pertanto, è importante introdurre in modo generale questo approccio per poi capire come è stato adottato nel contesto specifico dell'identificazione dei code smells.

L'elaborazione del linguaggio naturale (NLP) è un campo dell'intelligenza artificiale che si occupa dell'interazione tra i computer e il linguaggio umano. Il suo obiettivo principale è consentire ai computer di comprendere, interpretare e generare il linguaggio naturale in modo simile a come lo fanno gli esseri umani [10].

Negli ultimi anni, i modelli di elaborazione del linguaggio naturale (NLP) hanno ottenuto risultati notevoli e hanno suscitato grande interesse nella comunità di ricerca. Grazie ai numerosi successi ottenuti, la ricerca si sta sempre più focalizzando sull'utilizzo di queste tecniche e tecnologie innovative [11] [12].

Un importante contributo è rappresentato dai transformers, che sono modelli di apprendimento automatico basati su reti neurali. I transformers si sono dimostrati straordinariamente efficaci nell'elaborazione del linguaggio naturale [13], superando le limitazioni dei modelli precedenti basati su reti neurali, in quanto, una caratteristica distintiva dei transformers è l'uso di meccanismi di attenzione per catturare le relazioni a lungo termine tra le parole o le parti di un testo [11].

Un'altra caratteristica fondamentale di questi modelli è la loro capacità di essere pre-addestrati su grandi quantità di dati non etichettati e successivamente adattati (fine-tuning) a compiti specifici utilizzando dati etichettati relativi a quel compito. Quest'operazione di pre-addestramento e di fine-tuning è conosciuta anche come transfer learning. Nello specifico, il transfer learning [14] è una potente tecnica nell'apprendimento automatico (machine learning) che consente di sfruttare le conoscenze acquisite da modelli preaddestrati su grandi set di dati per migliorare

l'addestramento e le prestazioni di modelli in nuovi problemi o domini correlati.

L'idea alla base del transfer learning è che i modelli preaddestrati su task o domini complessi hanno imparato a riconoscere caratteristiche generali e astratte che possono essere utili anche in altri contesti [14]. Ad esempio, un modello addestrato su un'enorme quantità di immagini potrebbe aver imparato a riconoscere forme, contorni, texture e altre caratteristiche visive fondamentali. Queste conoscenze generali possono essere applicate con successo a nuovi problemi che richiedono l'elaborazione di immagini, anche se il set di dati di destinazione è molto più limitato.

Il processo di transfer learning comporta solitamente due fasi principali: il preaddestramento e il fine-tuning [14]. Nel preaddestramento, un modello viene addestrato su un grande dataset per estrarre caratteristiche rilevanti. Questa fase richiede un notevole dispendio di risorse computazionali e temporali [15], ma può essere eseguita in modo indipendente e in parallelo rispetto al problema specifico che si intende risolvere.

Nella fase di fine-tuning [16], il modello preaddestrato viene adattato o esteso specificamente per il compito in questione. Durante il fine-tuning, i pesi del modello vengono aggiornati in base al dominio di destinazione [16], consentendo al modello di apprendere informazioni più specifiche e di adattarsi al nuovo contesto.

Il transfer learning offre numerosi vantaggi. Innanzitutto, riduce la necessità di grandi quantità di dati di addestramento nel dominio di destinazione, poiché il modello iniziale ha già appreso caratteristiche generali, ciò è particolarmente utile quando si dispone di un insieme di dati limitato o costoso da raccogliere. Inoltre, il transfer learning accelera l'iter di sviluppo dei modelli, poiché si può partire da un modello di partenza già addestrato. Infine, può migliorare le prestazioni complessive del modello, poiché le conoscenze apprese nel dominio sorgente possono essere utilizzate per affrontare problemi simili nel dominio di destinazione [14].

Nel contesto specifico del rilevamento dei Code Smells, il transfer learning può offrire nuove prospettive per migliorare l'efficacia dei modelli utilizzati per identificare i problemi nel codice sorgente dei progetti software. Pertanto, la ricerca si concentra sull'utilizzo di approcci basati su NLP per identificare i code smell. Recentemente, sono stati condotti studi approfonditi [9] che confrontano attentamente le tecniche basate su euristiche, le tecniche di machine learning tradizionale e le tecniche di

NLP. Questa ricerca ha introdotto una nuova tecnica che ha portato a un'innovazione significativa.

Come dimostrato da diversi studi [9] [2] [3], l'identificazione dei code smell non è sempre una pratica semplice utilizzando tecniche tradizionali o basate su machine learning.

Lo studio condotto da Slivka et al. [9] rappresenta uno dei primi tentativi di valutare l'efficacia di tecniche di Deep Learning e di NLP per il rilevamento dei code smell. Negli ultimi tempi, si è assistito a un'avanzata significativa nell'elaborazione del linguaggio naturale (NLP) grazie all'introduzione delle rappresentazioni vettoriali delle parole come il BERT [11], Code2Vec e Code2Seq. Queste rappresentazioni catturano informazioni statistiche e semantiche del testo ed sono state dimostrate efficaci in vari compiti di NLP.

L'analisi condotta in questo studio è di fondamentale importanza, poiché ha esaminato attentamente l'uso degli embeddings e dei modelli di NLP preaddestrati. Sulla base di tali analisi, sono state acquisite conoscenze che hanno permesso di condurre uno studio approfondito sull'utilizzo delle tecniche di NLP per identificare i code smell.

Come menzionato in precedenza l'elaborazione del linguaggio naturale (NLP) ha trovato applicazioni innovative nel campo dell'ingegneria del software, anche nell'identificazione dei code smells [9]. L'utilizzo di tecniche di NLP per identificare i code smells può automatizzare e semplificare notevolmente il processo di analisi del codice, attraverso l'estrazione automatica delle caratteristiche linguistiche dal codice sorgente e attraverso l'analisi del contesto identificando le relazioni tra le frasi [11].

L'elaborazione del linguaggio naturale offre un approccio automatizzato per identificare i code smells analizzando il testo del codice sorgente [9]. Le tecniche di NLP possono essere utilizzate per estrarre informazioni semantiche, relazioni e pattern dal codice, al fine di identificare i code smells in modo più accurato ed efficiente [11] [9]. Questo può essere realizzato utilizzando algoritmi di apprendimento automatico, come le reti neurali, addestrate su un ampio corpus di codice sorgente.
I vantaggi dell'utilizzo di NLP per l'identificazione dei code smells sono molteplici, ad esempio:

1. Automazione: Così come le due tecniche citate in precedenza, l'utilizzo di NLP automatizza il processo di identificazione dei code smells, riducendo l'onere dell'analisi manuale del codice [9].

2. Precisione: Le tecniche di NLP possono catturare dettagli sottili nel codice sorgente, consentendo di identificare code smells che potrebbero essere sfuggiti all'occhio umano [9].

3. Scalabilità: L'elaborazione automatica del linguaggio può essere applicata a grandi quantità di codice, rendendo possibile l'identificazione dei code smells su progetti di grandi dimensioni [9].

CAPITOLO 3

Strategia di ricerca

3.1 Overview dell'approccio di ricerca

Questo capitolo presenta le scelte di progettazione e l'implementazione di vari modelli, partendo da quelli basati sul Machine Learning tradizionale fino ad arrivare a quelli basati su Natural Language Processing (NLP). L'obiettivo principale è confrontare l'utilizzo di diverse tecniche per l'individuazione dei Code Smells e fornire una valutazione preliminare sull'efficacia del transfer learning come possibile alternativa per la rilevazione di code smell.

La prospettiva adottata in questa ricerca è sia quella degli accademici interessati a riconoscere le limitazioni dei metodi esistenti, sia quella dei professionisti desiderosi di valutare l'applicabilità dell'apprendimento automatico per la rilevazione dei Code Smells. La domanda alla quale si cerca risposta in questa ricerca è:

RQ$_1$. *Le tecniche di NLP per identificare code smell sono migliori rispetto alle tradizionali tecniche di ML?*

3.2 Contesto applicativo

Il contesto di questo studio si basa su due fattori chiave: i progetti analizzati e la presenza o assenza di Code Smells. Per condurre la ricerca, è stato utilizzato un dataset pubblico che comprendeva 15 progetti open-source in Java disponibili su GitHub. Per evitare possibili distorsioni nella fase di addestramento dei modelli derivanti dall'utilizzo di dati provenienti da versioni precedenti dei progetti, è stato selezionato una singola versione di rilascio per ciascun progetto. La Tabella 3.1 riporta le caratteristiche principali di ciascun software incluso nello studio.
Il dataset utilizzato contiene dati relativi ai 15 progetti presi in considerazione nel nostro studio e comprende sei tipi di Code Smell: Complex Class, Large Class, Spaghetti Code, Class Data Should Be Private, Refused Bequest e Lazy Class. Nella Tabella 3.2 viene fornita una descrizione di ciascuno di questi code smell. La scelta di focalizzarsi su questi specifici tipi di smell è stata guidata principalmente dal fatto che sono stati ampiamente studiati dalla comunità di ricerca [2] [3], e sono considerati tra i più significativi e dannosi per gli sviluppatori, in quanto possono avere un impatto significativo sulla manutenibilità del codice sorgente [2] [3] [9].

Tabella 3.1: Progetti considerati nel contesto dello studio

Project Name	Release Tag	Classes	LOC
ant	rel/1.8.3	1163	169 510
ant-ivy	2.0.0-alpha2	388	36 665
cassandra cassandra-1.0.0 644 68 160			
elasticsearch	v0.19.0	2327	188 689
hadoop	release-0.6.0	297	40 129
hive	release-0.9.0	1268	161 239
hsqldb	2.2.8	589	182 898
karaf	karaf-2.3.0	549	42 420
lucene	releases/lucene-solr/3.6.0	3608	406 298
manifold-cf	release-0.6	729	119 998
nutch	release-1.4	296	30 024
pig	release-0.8.0	1345	206 603
qpid	0.14	1528	150 037
struts	STRUTS_2_3_4	1797	148 151
xerces2-j	Xerces-J_2_3_8	771	113 385

Tabella 3.2: Code Smell considerati nel contesto dello studio

Name	Description
Complex Class	Una classe complessa è caratterizzata da un elevato livello di complessità, con troppe responsabilità e un eccessivo numero di metodi e attributi. Questo rende difficile la comprensione e la manutenzione del codice.
Large Class	Una classe grande presenta un'elevata quantità di codice e spesso svolge molte responsabilità diverse. Questo può rendere il codice difficile da gestire, comprensibile e modificare in modo efficiente.
Lazy Class	Una classe pigra è una classe che non fornisce abbastanza funzionalità o valore per giustificarne la presenza nel sistema. Queste classi spesso non vengono utilizzate o hanno un impatto minimo sulle funzionalità complessive del software.
Class Data Should Be Private	Questo code smell si riferisce al fatto che i dati di una classe dovrebbero essere dichiarati come privati, ma invece sono accessibili pubblicamente o protetti. Ciò può portare a problemi di incapsulamento e alla violazione del principio di information hiding.
Refused Bequest	Questo code smell si verifica quando una classe eredita da un'altra classe ma non utilizza o riscrive la maggior parte dei metodi ereditati. Questo può indicare un'errata gerarchia di classi o un cattivo utilizzo dell'ereditarietà.
Spaghetti Code	Lo Spaghetti Code (codice spaghetti) si riferisce a un insieme di codice sorgente complesso e confuso, in cui le dipendenze tra le componenti non sono chiare e i flussi di esecuzione sono intricati. Questo rende difficile la comprensione, la manutenzione e l'estensione del codice.

3.3 Setup

3.3.1 Panoramica generale

In questo capitolo, forniremo una panoramica dettagliata del processo che è stato seguito per condurre la nostra ricerca sui Code Smell, al fine di ottenere risultati significativi. La pipeline di lavoro si articola principalmente in cinque fasi chiave:

1. Creazione del dataset finale: In questa fase iniziale, ci si è focalizzati sulla raccolta dei dati necessari per la ricerca. Sono stati selezionati i progetti pertinenti ed è stata eseguita un'analisi accurata per identificare le metriche del software utili come predittori. Il passo successivo è stato quello di creare un dataset in cui ogni riga rappresentava una classe appartenente ad un progetto, ogni colonna conteneva il valore di una metrica calcolata per quella classe, ed infine è stata aggiunta un'ultima colonna contenente il codice sorgente della classe Java.

2. Preparazione dei dati: Dopo aver creato il dataset finale, è stata eseguita una fase di pulizia per garantire la qualità dei dati. Sono state identificate e affrontate eventuali anomalie, ad esempio: i valori mancanti. Di conseguenza sono state adottate tecniche di gestione dei dati mancanti per garantire che il dataset fosse affidabile e coerente. La pulizia del dataset è fondamentale per evitare distorsioni o risultati inaccurati durante la fase di addestramento dei modelli.

3. Scelta dei modelli di Machine Learning tradizionale e modelli di Deep Learning: Durante la terza fase del nostro studio, abbiamo affrontato la questione della selezione dei modelli da utilizzare per l'analisi dei Code Smell. Abbiamo preso in considerazione sia i modelli di Machine Learning tradizionale che i modelli di Deep Learning, concentrandoci principalmente sui modelli di NLP al fine di coprire diverse prospettive e approcci. Di seguito sono presentati i modelli che abbiamo selezionato per il nostro studio: RandomForest, RandomForest con embeddings del codice sorgente, ed infine CodeBERT.
   Abbiamo selezionato attentamente questi modelli in base al contesto della nostra ricerca, tenendo conto delle peculiarità dei Code Smell e delle esigenze specifiche del nostro studio. L'obiettivo era garantire che i modelli fossero in

grado di affrontare le sfide legate all'identificazione dei Code Smell e di fornire risultati accurati e utili per la nostra analisi.

4. Addestramento dei modelli: Durante la quarta fase, ci si è concentrati sull'addestramento corretto dei modelli selezionati. Poiché i modelli differiscono tra loro, è stato importante fornire loro dati nel formato adeguato. Ad esempio, il modello CodeBERT richiede in input le classi Java tokenizzate, mentre RandomForest richiede le metriche come input. Pertanto, abbiamo suddiviso correttamente le colonne dei dati per addestrare i modelli nel modo appropriato.
   È importante sottolineare che la nostra ricerca si è focalizzata sulla classificazione "with-in project". Di conseguenza, i tre modelli sono stati addestrati sullo stesso training set e testati sullo stesso test set. Questo ci ha permesso di confrontare le prestazioni dei modelli in modo significativo e coerenente.

5. Metriche di valutazione: per valutare i risultati ottenuti dai vari modelli, sono state selezionate diverse metriche di valutazione, oltre a test statistici appropriati per confrontare e valutare i risultati dei modelli. L'obiettivo di questi test è stato valutare l'efficacia delle soluzioni proposte e determinarne la rilevanza nel contesto dei Code Smell. Sono state prese in considerazione diverse misure e indicatori per valutare le performance dei modelli e ottenere una visione completa delle loro prestazioni. Per questo studio le metriche utilizzate sono state: precision, recall, accuracy, f1, mcc, i test di Friedman, McNemar ed infine, la correzione di Bonferroni.

Complessivamente, la pipeline di lavoro è stata progettata in modo rigoroso per garantire un'analisi accurata e significativa dei Code Smell. Le fasi descritte sopra hanno fornito un quadro completo delle attività svolte durante la ricerca, assicurando l'affidabilità e la validità dei risultati ottenuti.

3.3.2 Creazione del dataset finale

Durante la fase iniziale del progetto, è stato dedicato un considerevole sforzo all'individuazione del dataset più idoneo. Il primo dataset identificato presentava alcune qualità positive, come ad esempio un'ampia quantità di dati e un vasto

insieme di snippet di codice Java classificati con precisione come "affetti" o "non affetti" da code smell [17]. Tuttavia, presentava alcune limitazioni come ad esempio la difficoltà nel reperire il codice sorgente necessario per eseguire la classificazione utilizzando modelli di NLP.

Di conseguenza, si è deciso di adottare un diverso dataset utilizzato in diversi studi [2] [3]. In questo nuovo dataset erano presenti le metriche relative ad ogni classe Java e il path per raggiungere le classi. Le metriche incluse nel dataset sono riportate nella Tabella 3.3.2. Una volta acquisiti i 15 progetti, è stato eseguito il processo di individuazione delle classi rilevanti mediante l'utilizzo di uno script Python. Tali classi sono state identificate grazie al path presente in questo dataset, e successivamente, sono state trasformate in stringhe e aggiunte al dataset finale, permettendo così di utilizzare il dataset sia per i modelli basati su NLP che per i modelli di machine learning tradizionali. Questa integrazione dei due approcci offre un'opportunità di confronto e una visione più completa nella classificazione dei code smell.

In conclusione, il processo di individuazione del dataset appropriato e la preparazione del dataset finale rappresentano fasi cruciali per consentire l'utilizzo congiunto di modelli basati su NLP e modelli di machine learning tradizionali nell'ambito dell'identificazione dei code smell.

Acronym	Full Name	Intrested Smell
Project_name	-	all
Component	-	all
CBO	Coupling Between Objects	Spaghetti Code
CYCLO	Cyclomatic Complexity	Spaghetti Code, Large Class
DIT	Depth of Inheritance Tree	Refused Bequest
ELOC	Executable Lines of Code	Spaghetti Code, Large Class
FanIn	Fan-In	-
FanOut	Fan-Out	-
LCOM	Lack of Cohesion of Methods	Large Class
LOC	Lines of Code	Large Class
LOCNAMM	Lines of Code Non-Accessor and Non-Mutator Methods	Large Class
NOA	Number of Attributes	Large Class
NOC	Number of Children	Complex Class
NOM	Number of Methods	Large Class
NOMNAMM	Number of Overridden and Non-Accessor Methods	Complex Class
NOPA	Number of Public Attributes	Class Data Should Be Private
PMMM	Percent Methods with More Methods	Complex Class
PRB	probability of refused bequest	Refused Bequest
WLOCNAMM	Weighted Lines of Code Non-Accessor and Non-Mutator Methods	Complex Class
WMC	Weighted Methods per Class	Complex Class
WMCNAMM	Weighted Methods per Class Non-Accessor and Non-Mutator Methods	Complex Class, Lazy Class
NMNOPARAM	Number of Methods with No Parameters	Spaghetti Code

3.3.3 Preparazione dei dati

La seconda fase della ricerca ha riguardato la preparazione dei dati, che rappresenta una delle fasi più cruciali e fondamentali dell'intero progetto. È di importanza vitale avere un dataset corretto e pronto all'uso per ottenere risultati accurati. È importante ricordare che i modelli utilizzati in questa ricerca non sono altro che modelli che effettuano inferenze su dati matematici, pertanto è essenziale garantire che tali dati siano il più chiari possibile.

Durante la fase di preparazione dei dati, sono state eseguite diverse attività per rendere il dataset adeguato all'utilizzo con i modelli di elaborazione del linguaggio naturale (NLP) e di machine learning. Queste attività comprendono la pulizia dei dati, l'eliminazione di informazioni non rilevanti o ridondanti, la normalizzazione e la codifica dei dati in un formato compatibile con i modelli utilizzati.

Questo processo di preparazione dei dati è fondamentale per ottenere risultati affidabili e significativi, consentendo ai modelli di NLP e di machine learning di lavorare su dati di qualità e di compiere inferenze accurate.

Di seguito sono state dettagliate le operazioni eseguite:

1. Identificazione e rimozione delle "stop words"

   Per determinare le parole di maggiore rilevanza all'interno di una frase al fine di comprenderne il significato, è comune eliminare le cosiddette stop-word. Le stop-word sono parole molto frequenti ma "riempitive" che non apportano informazioni rilevanti per la comprensione del significato della frase. Nel contesto di questa ricerca, l'eliminazione delle stop-word avviene sulla colonna "Component" del dataset, la quale rappresenta le calssi Java in formato stringa. L'eliminazione delle stop-word da classi Java implica la rimozione di elementi come commenti su singola riga, commenti multi-riga, import dei pacchetti, definizione del package, nonché la rimozione di caratteri come "\n" e "\t".

   Questa operazione di eliminazione delle stop-word si rivela utile per ridurre il rumore nei dati in esame e concentrarsi sulle parole chiave all'interno delle

classi Java, ottenendo così una rappresentazione più pulita e focalizzata sulla sostanza delle frasi.

In definitiva, l'eliminazione delle stop-word rappresenta una strategia efficace per migliorare l'analisi delle frasi, ridurre il rumore e mettere in evidenza le parole chiave all'interno delle classi Java in formato stringa.

2. Pulizia dei dati

   Durante questa fase, è stata condotta un'analisi accurata del dataset per valutare l'integrità dei dati presenti. Tra le diverse problematiche che possono emergere in questa fase, una delle principali è rappresentata dalla presenza di valori nulli o, nel caso specifico, di valori NaN (Not a Number). Dal momento che i modelli di machine learning selezionati operano principalmente su dati di tipo numerico, la presenza di tali valori costituiva un problema significativo.

   L'approccio adottato per gestire questa situazione è stato il seguente: innanzitutto, sono state conteggiate le istanze contenenti valori NaN per ottenere una stima iniziale della loro frequenza. Dopo aver verificato che su un dataset composto da circa 8200 istanze, solo 245 di esse presentavano valori NaN, e, in particolare, tali valori erano sempre associati alla colonna "WLOCNAMM", sono state valutate due alternative possibili.

   La prima opzione considerata era l'eliminazione dell'intera colonna "WLOCNAMM". Tuttavia, questa strategia avrebbe comportato una perdita di informazioni troppo significativa, considerando l'importanza dei dati contenuti in quella colonna.

   Di conseguenza, è stata adottata la seconda opzione, che prevedeva l'eliminazione delle sole istanze che presentavano valori NaN nella colonna "WLOCNAMM". È stato tenuto presente che questa operazione avrebbe comportato una perdita di informazioni, seppur parziale.

   Questa decisione è stata presa in considerazione dopo un'attenta valutazione dei compromessi tra la necessità di mantenere l'integrità dei dati e l'impatto sulla quantità di informazioni perse. È stato riconosciuto che l'eliminazione delle istanze con valori NaN avrebbe comportato una riduzione delle dimensioni

del dataset e potenzialmente una riduzione della rappresentatività statistica, ma è stata considerata una scelta ragionevole per garantire la coerenza dei dati rimanenti.

In conclusione, durante questa fase di analisi dei dati, sono state affrontate e gestite le problematiche legate alla presenza di valori NaN nel dataset. La decisione di eliminare le istanze con valori NaN nella colonna "WLOCNAMM" è stata presa per preservare l'integrità generale del dataset, seppur a costo di una parziale perdita di informazioni. È importante sottolineare che questa scelta è stata valutata attentamente considerando il trade-off tra l'integrità dei dati e l'impatto sulla quantità di informazioni perse.

3. Normalizzazione dei dati

   Al fine di ottenere una classificazione più accurata, è stato necessario eseguire una normalizzazione dei dati presenti nel dataset. L'osservazione iniziale del dataset ha rivelato che molti attributi, in particolare le metriche relative alle diverse classi Java, presentano valori molto diversificati, il che potrebbe causare problemi al modello durante la fase di classificazione. Per affrontare questa problematica, è stata applicata la tecnica di normalizzazione MinMax.

   La normalizzazione MinMax viene eseguita su tutti gli attributi numerici del dataset, ad eccezione delle variabili dipendenti che rappresentano le classi Java da classificare. Questa tecnica di normalizzazione scala i valori degli attributi in un nuovo intervallo compreso tra 0 e 1. Ciò significa che il valore più basso dell'attributo viene mappato a 0, mentre il valore più alto viene mappato a 1. Gli altri valori vengono opportunamente scalati in base alla loro posizione all'interno dell'intervallo originale.

   L'output di questa trasformazione è un nuovo dataframe in cui tutti gli attributi numerici sono stati normalizzati nell'intervallo [0-1]. Questo processo garantisce che tutte le feature abbiano la stessa scala, evitando disparità di importanza tra gli attributi che potrebbero influenzare negativamente il modello di classificazione.

4. Bilanciamento del dataset

La ricerca si è concentrata su un dataset altamente sbilanciato, con un numero significativamente maggiore di istanze senza Code Smell rispetto a quelle con Code Smell, come mostrato nelle figure 3.1, 3.2, 3.3, 3.4, 3.5, 3.6. Tuttavia, non sono state adottate tecniche di bilanciamento del dataset. Alcune ricerche precedenti, come il lavoro condotto da Pecorelli et al. [3], hanno dimostrato che il bilanciamento del dataset non ha prodotto risultati significativi. In particolare, l'algoritmo di bilanciamento SMOTE utilizzato in tali studi non ha considerato uno spazio delle caratteristiche sufficientemente ampio per ottenere un bilanciamento efficace, risultando in un fallimento nell'ottenere risultati validi. Questi fallimenti non sono stati presi in considerazione nel calcolo delle metriche di valutazione utilizzate per valutare le prestazioni dei modelli. Pertanto, l'interpretazione dei risultati sarebbe stata influenzata dalla mancanza di molte previsioni valide, rendendo necessaria un'attenta e consapevole valutazione dei risultati ottenuti.

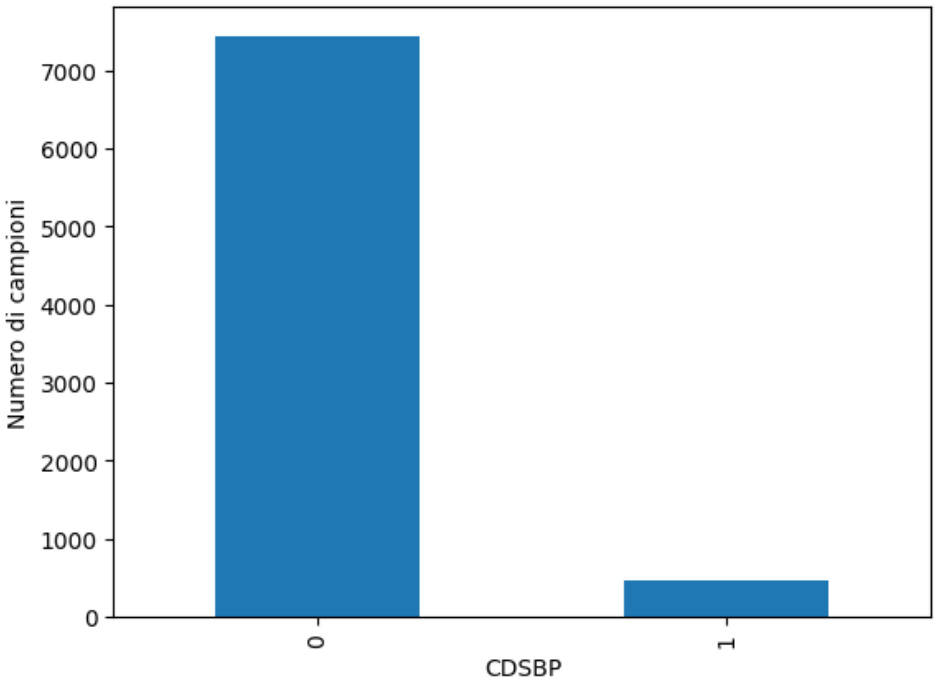

Figura 3.1: Class Data Should Be Private

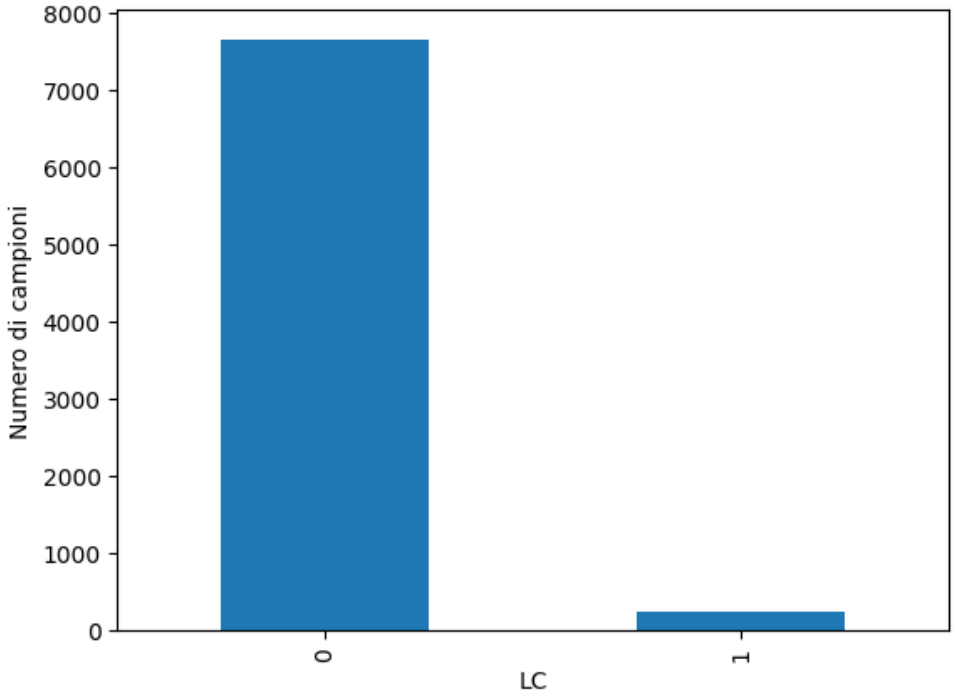

Figura 3.2: Long Class

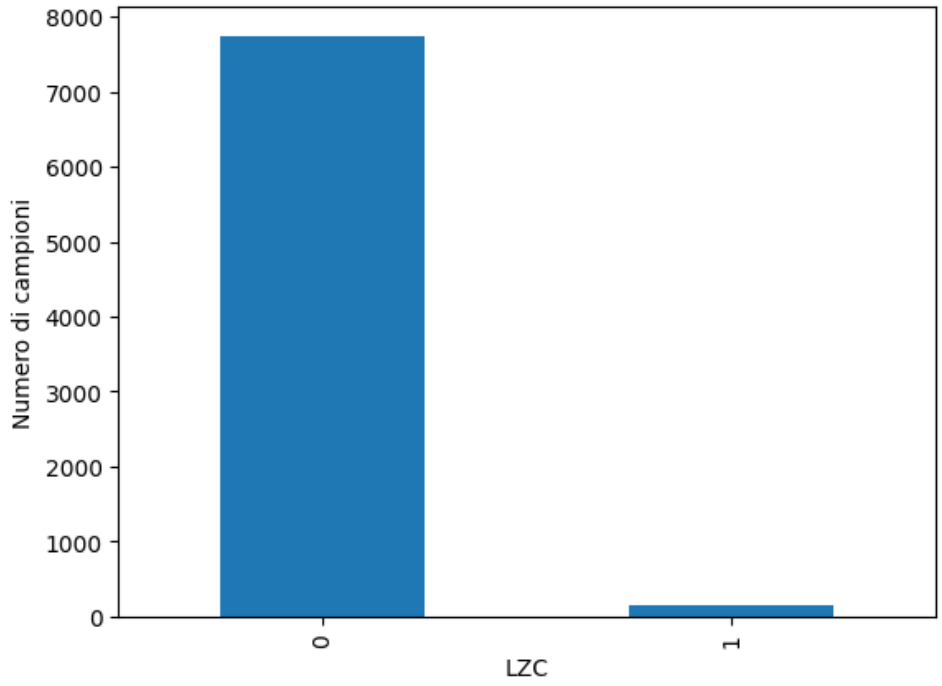

Figura 3.3: Lazy Class

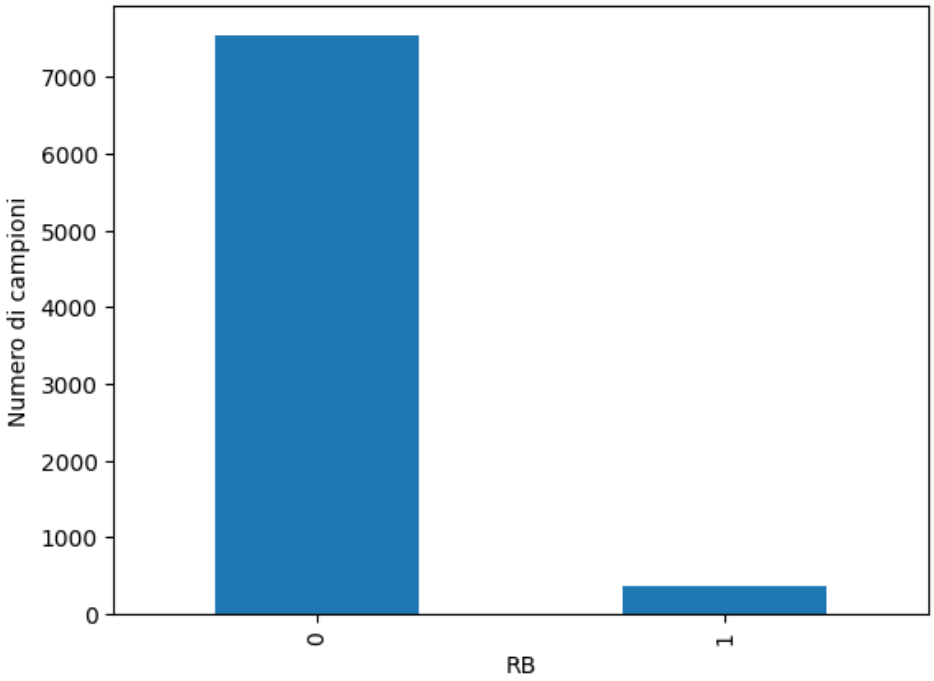

Figura 3.4: Refused Bequest

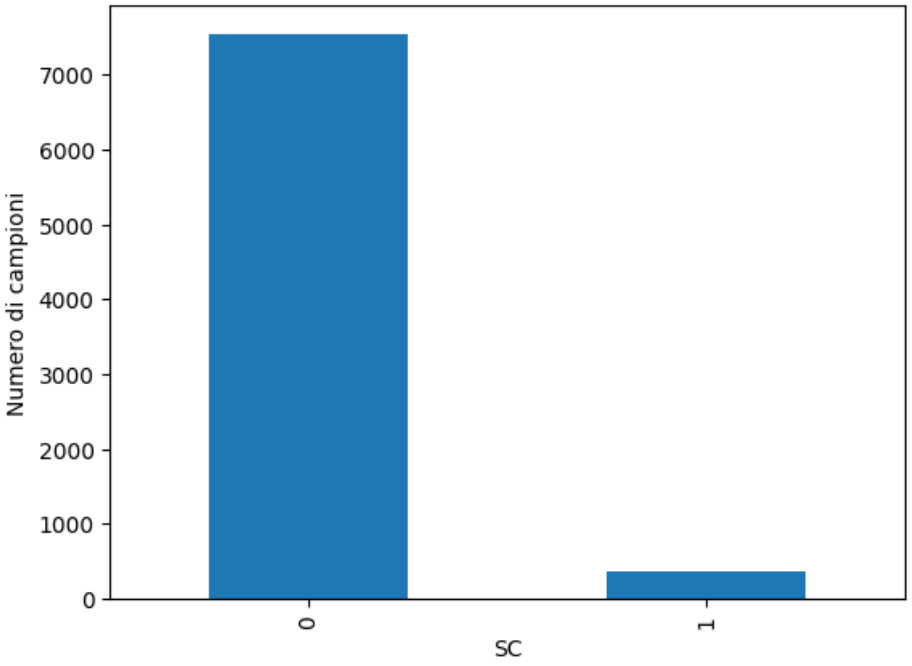

Figura 3.5: Spaghetti Code

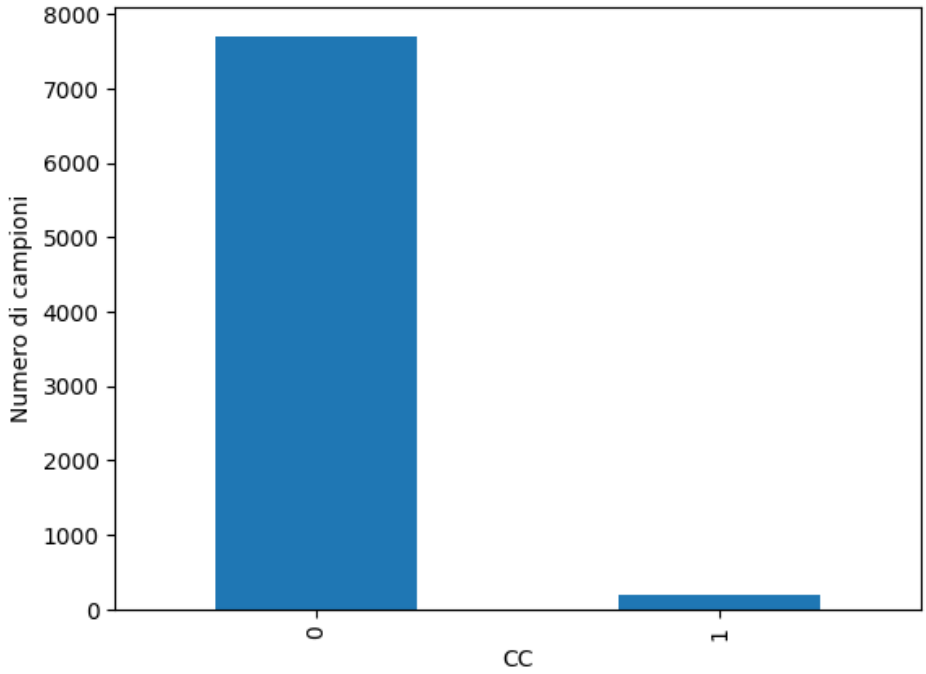

Figura 3.6: Complex Class

3.3.4 Scelta dei modelli di Machine Learning tradizionale e modelli di NLP

RandomForest La scelta di utilizzare un modello di RandomForest per la classificazione di code smells all'interno di un progetto presenta diversi vantaggi rispetto ad altri modelli. Uno dei principali vantaggi è la capacità di gestire in modo efficiente una grande quantità di dati e variabili. RandomForest è un algoritmo ensemble che combina molteplici alberi decisionali, ognuno dei quali viene addestrato su un sottoinsieme casuale del dataset di addestramento. Questo approccio riduce il rischio di overfitting, cioè di adattarsi eccessivamente ai dati di addestramento, e permette al modello di generalizzare bene su nuovi dati.

La natura ensemble di RandomForest consente di ottenere una migliore robustezza rispetto ad altri modelli. Poiché l'output finale del modello è una combinazione delle previsioni di molti alberi decisionali, gli errori compiuti da singoli alberi possono essere compensati da altri alberi. Ciò rende il modello meno sensibile a variazioni casuali o rumore nei dati di addestramento, migliorando la sua capacità di generalizzazione.

Un altro vantaggio di RandomForest è la sua capacità intrinseca di valutare l'importanza delle variabili nel processo decisionale. Durante la costruzione dei singoli alberi, RandomForest valuta quali variabili sono le più informative per separare le classi di interesse. Questa informazione sull'importanza delle variabili può essere utilizzata per identificare i code smells più significativi all'interno di un progetto.

Infine, la scelta di utilizzare un modello di RandomForest per la classificazione dei code smells all'interno del progetto è stata ulteriormente supportata dall'accuratezza riscontrata nelle prove condotte nello studio di De Stefano et al. [2]. Questo studio evidenzia le prestazioni discrete del modello di RandomForest nella classificazione dei code smells all'interno di singoli progetti.

Dall'analisi dettagliata dei risultati ottenuti, emerge che il modello RandomForest ha dimostrato una performance superiore rispetto agli altri classificatori nell'identificazione dei code smell. Nonostante non sia eccellente, risulta essere il miglior modello tra quelli valutati.

Pertanto, sulla base delle evidenze fornite nel paper [2], è stato possibile fare una scelta informata e consapevole nel selezionare il modello di RandomForest per la classificazione dei code smells all'interno del progetto.

RandomForest con word2vec Come mostrato in altri studi [9], l'uso di embeddings potrebbe migliorare le capacità predittive di modelli di machine learning tradizionale. Per la vettorizzazione delle classi Java, è stato scelto di adottare l'approccio basato su Word2Vec [18], poiché rappresenta una soluzione potente e innovativa per catturare la rappresentazione semantica delle parole. Word2Vec è una tecnica di word embedding, che permette di convertire parole o frasi in vettori numerici, preservando le relazioni semantiche tra di esse [18].

L'idea fondamentale di Word2Vec è che il significato di una parola può essere catturato attraverso l'analisi del contesto in cui appare. L'algoritmo considera una finestra di contesto intorno a ciascuna parola all'interno di un testo e apprende ad associare parole simili con vettori simili. Pertanto, parole che condividono contesti simili avranno rappresentazioni vettoriali più simili [18].

L'applicazione di Word2Vec a questo contesto potrebbe consentire di acquisire una migliore comprensione delle caratteristiche semantiche delle classi Java. I

vettori generati da Word2Vec possono essere utilizzati come caratteristiche aggiuntive o sostitutive per le classi Java nel processo di addestramento del modello di classificazione.

Gli embeddings generati da Word2Vec catturano le informazioni semantiche delle classi Java. Passare questi embeddings alla Random Forest consente al modello di apprendere le relazioni semantiche tra le classi in modo più efficace rispetto a una rappresentazione basata su caratteristiche tradizionali. Ciò può portare a una migliore comprensione dei dati e migliorare le previsioni della Random Forest.

CodeBert CodeBERT[12] è un modello di linguaggio pre-addestrato specificamente progettato per l'elaborazione del codice sorgente.

Basato sull'architettura dei trasformatori di BERT[11], CodeBERT apprende rappresentazioni contestualizzate del codice sorgente. Queste rappresentazioni catturano sia le informazioni sintattiche che semantiche del codice, consentendo al modello di comprendere e generare codice sorgente in modo più accurato.

Con CodeBERT, è possibile affrontare una serie di compiti relativi al codice sorgente, come il completamento automatico del codice, la correzione degli errori, o, in generale, l'estrazione delle informazioni dal codice. Il modello può essere utilizzato sia per compiti supervisionati, quando si dispone di un set di dati etichettato, sia per compiti di auto-apprendimento, in cui il modello può essere adattato a dati non etichettati tramite il processo di fine-tuning.

CodeBERT è stato reperito attraverso Hugging Face, una piattaforma che mette a disposizione un vasto numero di modelli pre-addestrati.

La scelta di questo modello non è stata semplice: prima di selezionare CodeBERT, sono stati eseguiti numerosi test su modelli simili, principalmente basati su BERT. Inizialmente, si è considerato BERT stesso, oppure una versione più leggera come ALBERT, o un modello con meno vincoli sull'input come longformer-base-4096.

La ragione principale per la scelta di CodeBERT sono state le sue prestazioni. Tuttavia, il modello presenta alcune limitazioni, come il limite sulla dimensione dell'input che può elaborare. Poiché stiamo lavorando con classi Java molto estese, è stato necessario effettuare un'attenta pulizia del testo, ma ciò non è stato sempre sufficiente. Data la limitata disponibilità di risorse computazionali, CodeBERT richiede

meno risorse rispetto agli altri modelli menzionati, sebbene comunque richieda una quantità significativa. Pertanto, la scelta del modello è stata effettuata considerando un trade-off tra "prestazioni" e "risorse computazionali". In conclusione era impossibile utilizzare modelli più performanti e meno vincolati con le strumentazioni a disposizione.

3.3.5 Addestramento dei modelli

Una volta selezionati i modelli, si è proceduto alla fase di addestramento. Il dataset è stato inizialmente suddiviso in un set di addestramento e un set di test, con quest'ultimo corrispondente al 33% del totale. L'approccio adottato in questo caso è conosciuto come "with-in project". Questo metodo viene utilizzato nel campo della classificazione dei dati per valutare le prestazioni dei modelli predittivi all'interno di un contesto specifico, attraverso la suddivisione del dataset in base ai progetti. L'obiettivo principale dell'approccio "with-in project" è analizzare il comportamento dei modelli all'interno di ciascun progetto, tenendo conto delle variazioni e delle peculiarità che possono essere presenti tra i diversi progetti.

Per implementare questo approccio, il dataset viene diviso in base ai progetti. Ad ogni passaggio al successivo progetto, il dataset viene partizionato in un training set e un test set specifico per quel progetto. In pratica, i modelli vengono addestrati su un determinato progetto e successivamente testati su quello stesso progetto, prima di procedere al prossimo.

Questa metodologia consente di valutare le prestazioni dei modelli in modo accurato, considerando le caratteristiche uniche di ciascun progetto. In tal modo, è possibile ottenere una visione più dettagliata delle capacità predittive dei modelli all'interno di contesti specifici.

Come accennato in precedenza, i modelli richiedono dati di input in formati diversi. Per adattarsi alle esigenze specifiche dei modelli CodeBERT e RandomForest con embeddings, sono state selezionate le colonne pertinenti dal set di addestramento. Per il modello CodeBERT, è stata estratta la colonna contenente il codice Java, ovvero la colonna denominata "Component" 3.3.2, mentre per la RandomForest tradizionale

sono state utilizzate solo le metriche 3.3.2, escludendo la colonna "Component" e la colonna "Project_name".

Dopo aver ottenuto la colonna con il codice sorgente, il passo successivo è stato la trasformazione dei dati in formati adatti ai modelli CodeBERT e RandomForest con embeddings. Il modello CodeBERT, essendo basato su BERT, richiede che i dati testuali siano convertiti in sequenze numeriche utilizzando un tokenizer specifico [11] [12]. Il tokenizer suddivide il testo in unità semanticamente rilevanti, come parole o sotto-parole, e assegna a ciascuna di esse una rappresentazione numerica univoca. Queste sequenze numeriche vengono quindi utilizzate come input per il modello CodeBERT.

Analogamente al modello CodeBERT, il modello RandomForest con embeddings richiede rappresentazioni numeriche delle parole nel codice sorgente. Per ottenere tali rappresentazioni, è stato utilizzato il modello Word2Vec [18], un modello di word embedding che cattura le relazioni semantiche tra le parole basandosi sul contesto in cui compaiono. Nel contesto di questo studio, è stato utilizzato un modello Word2Vec di tipo "Continuous Bag of Words" (CBOW) [19]. Nel modello CBOW, l'obiettivo è prevedere una parola di input a partire dal contesto circostante. Questa tecnica è stata scelta per catturare l'informazione di contesto delle parole [19].

Per quanto riguarda il modello RandomForest tradizionale, è stato addestrato utilizzando le metriche software menzionate nella Tabella 3.3.2. Per consentire l'utilizzo di tali dati come input per il modello, si è prestata attenzione a rimuovere eventuali valori NaN e ad assicurarsi che i dati fossero tutti nel formato numerico supportato.

3.3.6 Metriche di valutazione

Per valutare le prestazioni ottenute dalle tecniche sperimentate, abbiamo adottato un approccio basato su diverse metriche ampiamente utilizzate nel campo del recupero delle informazioni. In particolare, abbiamo considerato cinque metriche principali: precision, recall, accuracy, F-measure e il coefficiente di correlazione di Matthews (MCC). Queste metriche ci hanno fornito una valutazione completa delle prestazioni dei modelli in termini di precisione nella classificazione dei Code Smell. In seguito viene fornita una descrizione delle metriche applicate:

- Precision: misura la proporzione di istanze correttamente classificate come positivi tra tutte le istanze classificate come positive.

$$Precision = \frac{TP}{TP + FP} \tag{3.3.1}$$

- Recall: misura la proporzione di istanze positive correttamente identificate rispetto al numero totale di istanze positive effettive presenti nel dataset.

$$Recall = \frac{TP}{TP + FN} \tag{3.3.2}$$

- Accuracy: rappresenta la percentuale complessiva di classificazioni corrette, includendo sia i veri positivi che i veri negativi.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.3.3}$$

- F-measure: combina la precision e la recall in un unico valore, fornendo una misura complessiva delle prestazioni del modello.

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.3.4}$$

- Coefficiente di correlazione di Matthews (MCC): è una misura di associazione tra le predizioni del modello e le etichette effettive. Considera tutti i quattro risultati della classificazione e fornisce un valore compreso tra -1 e 1, indicando l'accuratezza complessiva del modello.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.3.5}$$

Come mostrato in molti articoli [20], i test statistici che possono confrontare i modelli basati su un singolo set di test sono diventati una considerazione importante nel campo del machine learning moderno, soprattutto per i modelli di deep learning.

I modelli di deep learning sono spesso caratterizzati da dimensioni notevoli e operano su set di dati molto ampi. Questi fattori possono comportare tempi di addestramento che richiedono giorni o settimane su hardware moderno e performante.

Questa complessità computazionale rende impraticabile l'uso dei metodi di ricampionamento, che richiedono la ripetizione dell'addestramento dei modelli su diverse partizioni dei dati. Pertanto, c'è la necessità di utilizzare un test che possa operare sui risultati della valutazione dei modelli addestrati su un singolo set di dati di test.

In questo contesto, il test di Friedman si configura come un'opzione adatta per valutare i modelli di deep learning, che spesso richiedono tempi di addestramento lunghi. Il test di Friedman è particolarmente utile per confrontare le prestazioni di più modelli su un set di dati di test comune.

Questo test si basa sulla classifica delle prestazioni dei modelli su diverse misurazioni o campioni. L'obiettivo del test è valutare se ci sono differenze significative tra le prestazioni complessive dei modelli. In altre parole, il test di Friedman verifica se i modelli hanno medie di prestazioni significativamente diverse.

Utilizzando il test di Friedman, siamo in grado di ottenere una valutazione più dettagliata e accurata delle differenze tra i modelli, consentendoci di identificare quali modelli presentano prestazioni significativamente diverse. Questo ci aiuta a prendere decisioni informate sulla scelta del modello migliore per le nostre esigenze specifiche, tenendo conto delle limitazioni computazionali e delle prestazioni generali dei modelli.

Per quanto riguarda l'applicazione del test di Friedman, è stato necessario inizialmente ottenere le predizioni dei tre modelli, in quanto il test si basa sul confronto delle differenze tra le loro predizioni. Dopo aver salvato queste informazioni in un file CSV, è stato possibile procedere con l'implementazione dello script Python per l'applicazione del test utilizzando librerie specifiche.

Per eseguire correttamente il test di Friedman, sono state utilizzate librerie come pandas e scipy in Python. La libreria pandas è stata utilizzata per caricare i dati dal file CSV in un dataframe, mentre la libreria scipy offre funzioni per il calcolo del test di Friedman.

In aggiunta al test di Friedman, si è ritenuto opportuno ricorrere a ulteriori test statistici al fine di analizzare con maggiore precisione le significative discrepanze tra i tre modelli considerati. Il test di Friedman, infatti, pur essendo uno strumento utile per fornire una visione d'insieme sulle disparità complessive tra i modelli, è

fondamentale sottolineare che esso non delinea in modo specifico quali coppie di modelli presentino tali discrepanze.

Al fine di approfondire l'analisi e fornire una comprensione più dettagliata delle discrepanze nelle classificazioni tra i modelli, è stato utilizzato il test di McNemar come test "post hoc". Questo test ha consentito di identificare specificamente tra quali modelli si verificavano le differenze significative, contribuendo a giustificare in modo più approfondito le tendenze osservate nei grafici.

Attraverso l'implementazione del test di McNemar, è stato possibile valutare le discrepanze nelle classificazioni tra coppie di modelli. Ciò ha permesso di identificare le situazioni in cui un modello mostrava prestazioni significativamente diverse rispetto agli altri due. L'analisi di queste disparità specifiche ha fornito una comprensione più approfondita delle ragioni per cui certi modelli avevano risultati migliori o peggiori in determinati contesti.

Ai p-value ottenuti dalle varie esecuzioni del test di McNemar, però, è stata applicata la correzione di Bonferroni per gestire il problema dei confronti multipli. Questa correzione è stata essenziale per mantenere un livello di significatività adeguato, garantendo che le differenze individuate tra i modelli fossero effettivamente significative e non il risultato del puro caso.

Complessivamente, l'applicazione del test di McNemar e la correzione di Bonferroni come test "post hoc" hanno arricchito l'analisi dei risultati ottenuti dal test di Friedman, offrendo un'analisi dettagliata delle differenze nelle classificazioni tra i modelli e confermando ciò che era intuitivamente rilevabile dai grafici di confronto.

CAPITOLO 4

Analisi dei risultati

4.1 Overview dell'analisi dei risultati

Per analizzare le prestazioni dei diversi modelli, sono state utilizzate diverse fonti di informazioni. Inizialmente, per visualizzare in modo più intuitivo le prestazioni dei tre modelli, sono stati creati dei diagrammi (4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12) che confrontano la distribuzione delle metriche MCC (Matthews Correlation Coefficient) e F1-score. Questi diagrammi mettono in evidenza le differenze di prestazione tra i modelli in base a queste specifiche metriche, consentendo una rapida comprensione delle performance relative dei modelli.

Successivamente, sono stati eseguiti test statistici come il test di Friedman, il test di McNemar con correzione di Bonferroni per valutare la significatività delle differenze tra i modelli [20, 21, 22]. I risultati di questi test sono stati riportati all'interno di specifiche tabelle (4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7), che mostrano le statistiche di valutazione per ogni tipo di smell in ogni progetto.

Attraverso queste diverse fonti di informazioni - tabelle delle metriche, risultati dei test statistici e diagrammi comparativi - è possibile ottenere una visione completa delle prestazioni dei vari modelli e trarre conclusioni significative sull'efficacia di ciascun modello nello specifico contesto di analisi degli smell dei progetti.

4.2 Analisi dei risultati

In questa sezione la ricerca si è concentrata sull'analisi dei risultati ottenuti dai diversi classificatori e sulla loro spiegazione.

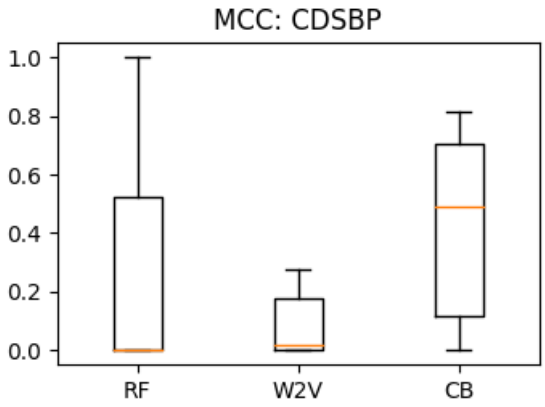

Figura 4.1: MCC: Class Data Should Be Private

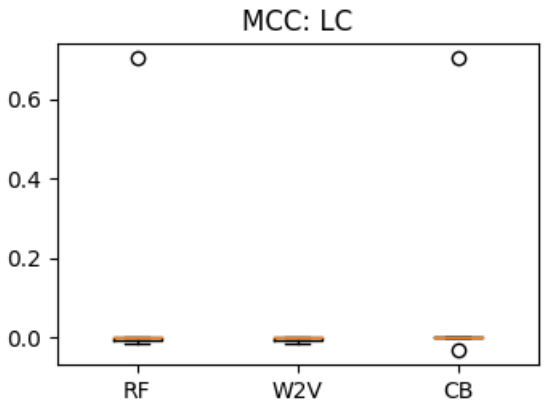

Figura 4.2: MCC: Large Class

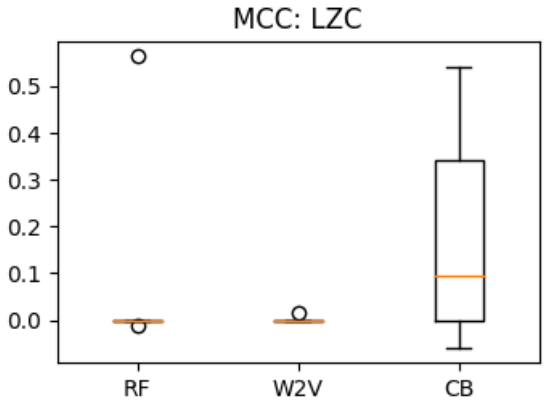

Figura 4.3: MCC: Lazy Class

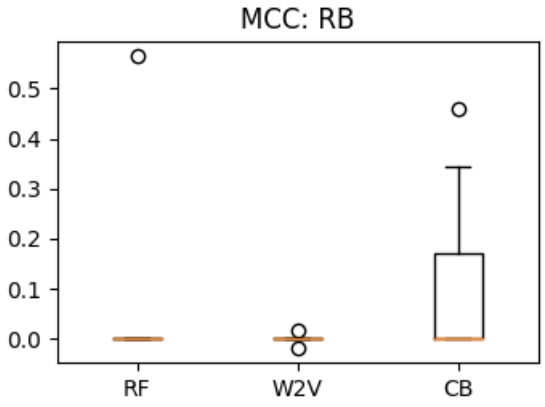

Figura 4.4: MCC: Refused Bequest

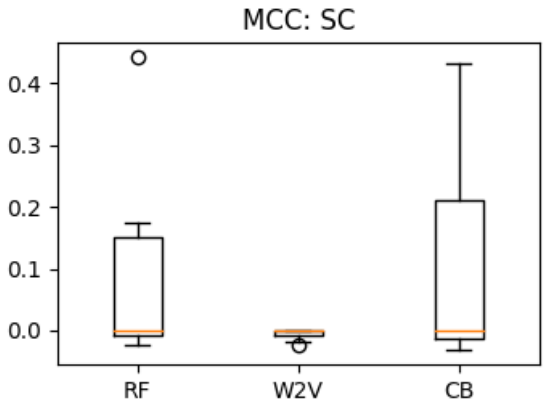

Figura 4.5: MCC: Spaghetti Code

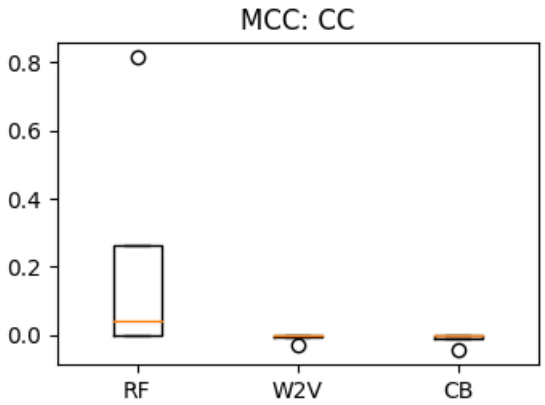

Figura 4.6: MCC: Complex Class

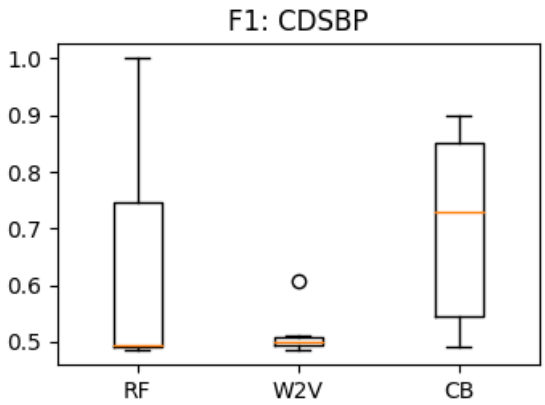

Figura 4.7: F1: Class Data Should Be Private

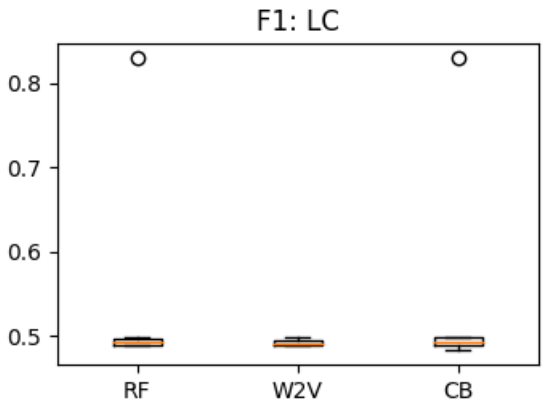

Figura 4.8: F1: Large Class

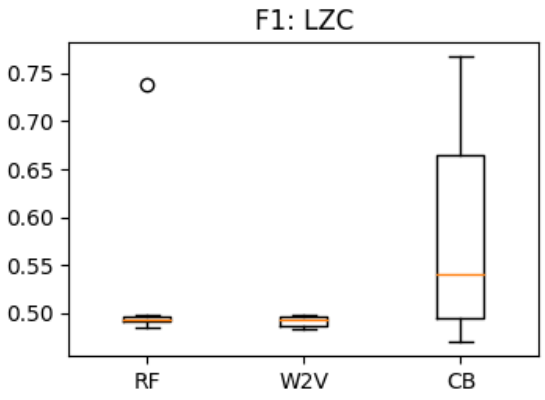

Figura 4.9: F1: Lazy Class

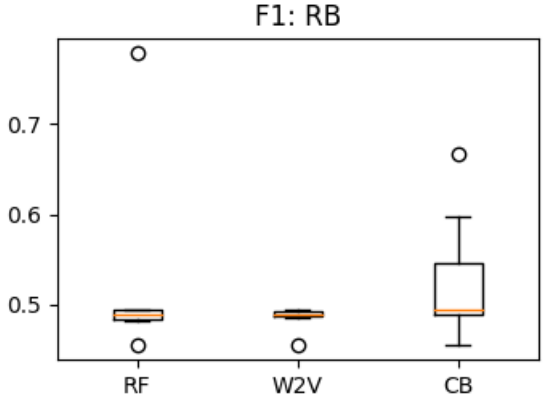

Figura 4.10: F1: Refused Bequest

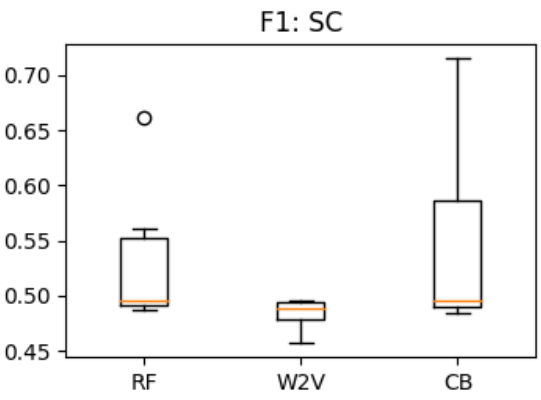

Figura 4.11: F1: Spaghetti Code

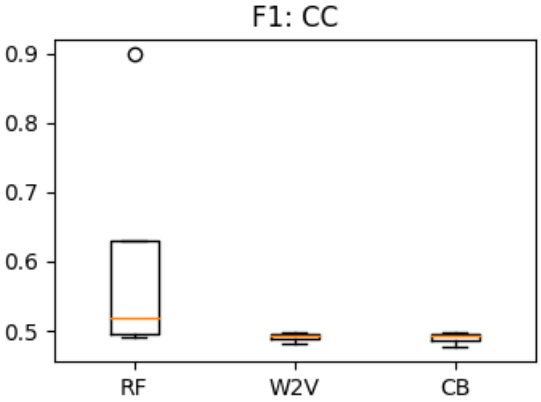

Figura 4.12: F1: Complex Class

Come prima considerazione, osservando i grafici 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12 è facile notare come questi mostrano un pattern comune: indipendentemente dal modello applicato, le prestazioni migliori si ottengono quando si classificano le istanze del code smell "Class Data Should Be Private", mentre le prestazioni peggiori si verificano per gli smell "Large Class". Questa conclusione è supportata dai grafici che mostrano la distribuzione delle metriche MCC e F1-score.

Esaminando il confronto tra i classificatori, come evidenziato nei grafici 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, emergono alcune tendenze. Sembra che il modello di NLP (CodeBERT) presenti le prestazioni migliori tra i tre classificatori considerati. Segue la RandomForest tradizionale e, infine, quella con embeddings.

Tabella 4.1: p-value ottenuti dal test di Friedman

Project Name	Smell	p-value
ant-rel-1.8.3	CDSBP	4.839314e-01
ant-rel-1.8.3	CC	NaN
ant-rel-1.8.3	RB	4.839314e-01
ant-rel-1.8.3	SC	5.087286e-03
cassandra-cassandra-1.0.0	CDSBP	4.771905e-01
cassandra-cassandra-1.0.0	LC	1.905601e-06
cassandra-cassandra-1.0.0	LZC	4.771905e-01
cassandra-cassandra-1.0.0	SC	4.075157e-03
elasticsearch-v0.19.0	LC	NaN
elasticsearch-v0.19.0	LZC	2.939117e-02
hadoop-release-0.6.0	LC	NaN
hadoop-release-0.6.0	LZC	4.719590e-01
hive-release-0.9.0	LC	3.319274e-12
hive-release-0.9.0	RB	NaN
hive-release-0.9.0	SC	Nan
hsqldb-2.2.8	CDSBP	4.803908e-01
hsqldb-2.2.8	LC	4.179316e-08
hsqldb-2.2.8	LZC	5.483436e-01
hsqldb-2.2.8	RB	0.00
hsqldb-2.2.8	SC	7.268251e-03

Tabella 4.2: p-value ottenuti dal test di Friedman

Project Name	Smell	p-value
lucene-releases-lucene-solr-3.6.0	CDSBP	6.021708e-02
lucene-releases-lucene-solr-3.6.0	CC	4.854463e-01
lucene-releases-lucene-solr-3.6.0	LZC	3.195215e-05
lucene-releases-lucene-solr-3.6.0	RB	NaN
lucene-releases-lucene-solr-3.6.0	SC	4.651498e-04
manifold-cf-release-0.6	RB	NaN
nutch-release-1.4	LZC	NaN
pig-release-0.8.0	CDSBP	NaN
pig-release-0.8.0	CC	NaN
pig-release-0.8.0	RB	5.968019e-04
pig-release-0.8.0	SC	NaN
xerces2-j-Xerces-J_{230}	CDSBP	1.210959e-06
xerces2-j-Xerces-J_{230}	CC	2.021236e-09
xerces2-j-Xerces-J_{230}	LC	NaN
xerces2-j-Xerces-J_{230}	RB	4.836299e-01
xerces2-j-Xerces-J_{230}	SC	3.154221e-06

Analizzando i vari grafici, si notano differenze nelle distribuzioni delle metriche tra i modelli. Tuttavia, per supportare queste affermazioni, sono stati applicati test statistici non parametrici, che hanno permesso di valutare le effettive discrepanze tra le predizioni dei tre modelli. Il primo test applicato è stato quello di Friedman, scelto in quanto i modelli da confrontare sono 3. I p-value ottenuti da questo test sono riportati nelle Tabelle 4.1, 4.2. Il test di Friedman ha mostrato, in più di un'occasione,

la presenza di una forte differenza nelle predizioni dei modelli. Poiché il test di Friedman permette di capire se tra i tre modelli ci sono differenze significative, ma non permette di identificare quali di essi presentano tali differenze, è stata applicata una tecnica di "post hoc test" utilizzando il test di McNemar. Questo test è stato utilizzato per identificare quale dei modelli presenta una differenza significativa nelle predizioni. I p-value ottenuti dal test di McNemar sono stati successivamente corretti utilizzando la correzione di Bonferroni per ridurre il rischio di errore di falsi positivi durante i molteplici confronti tra gruppi o condizioni 4.3, 4.4, 4.5, 4.6, 4.7.

Di seguito viene presentata un'analisi e un confronto più approfondito tra i tre modelli riguardo alla classificazione di ciascun code smell.

4.2.1 Risultati per Class Data Should Be Private

Come già menzionato in precedenza, l'analisi dei grafici 4.1 e 4.7 rivela che i modelli sono stati in grado di classificare correttamente lo smell di "Class Data Should Be Private". Tra i vari modelli utilizzati per l'identificazione dei code smell, CodeBERT ha dimostrato una discreta capacità nell'identificare tale smell. Successivamente, vi è il modello basato su machine learning tradizionale, seguito dal modello di RandomForest con embeddings.

Analizzando la distribuzione relativa alla metrica "F1" (Figura 4.7), possiamo osservare che le distribuzioni dei modelli si trovano al di sopra del 55%, raggiungendo un massimo dell'85% per il modello di NLP, mentre per la RandomForest tradizionale, la distribuzione oscilla tra il 50% e il 75%. Questi valori indicano che i modelli stanno classificando correttamente una percentuale significativa dei casi positivi, ma vi è ancora spazio per migliorare la capacità di identificarne altri. Il modello di RandomForest con embeddings invece presenta una distribuzione pari al 50%.

In particolare, il modello di NLP ha raggiunto prestazioni superiori con un valore massimo dell'F1-score pari all'85%. Ciò suggerisce che il modello ha dimostrato un buona capacità di bilanciare precision e recall nelle sue predizioni, identificando correttamente la maggior parte dei casi positivi.

D'altra parte, gli altri due modelli (RandomForest tradizionale e il modello RandomForest con embeddings) hanno ottenuto una distribuzione della F1-score leggermente inferiore. Questi valori suggeriscono anche che vi sono ancora alcune istanze positive che potrebbero essere migliorate nella classificazione.

Globalmente, i risultati relativi alla distribuzione della "F1" (Figura 4.7) indicano che i modelli di NLP e di machine learning tradizionale hanno prestazioni ragionevolmente buone, ma potrebbero beneficiare di ulteriori sforzi per aumentare l'accuratezza e l'affidabilità delle previsioni, specialmente nell'identificazione di casi positivi non ancora correttamente classificati.

Tabella 4.3: p-value relativi a CDSBP dopo aver effettuato la correzione di Bonferroni su McNemar

Project Name	Smell	RF vs. W2V	RF vs. CB	W2V vs. CB
xerces2-j-Xerces-J$_{230}$	CDSBP	2.771117e-13	6.000000	5.435652e-13

I grafici 4.1 e 4.7 mostrano che le predizioni dei modelli presentano qualche differenza tra loro, suggerendo che le prestazioni dei modelli potrebbero essere simili in generale. Tuttavia, per una comprensione più approfondita delle possibili differenze tra i modelli, abbiamo condotto il test di Friedman 4.1, 4.2. Questo test ha rivelato che, in un solo progetto, vi è una differenza significativa nelle predizioni dei modelli, indicando che i modelli potrebbero avere prestazioni diverse in circostanze specifiche.

Al fine di identificare tra quali modelli si verificano queste discrepanze significative, abbiamo applicato un ulteriore test statistico, quello di McNemar. Questo ci ha permesso di confrontare le prestazioni dei modelli due a due e individuare quali presentano differenze significative nelle loro previsioni.

Il test di McNemar ha evidenziato una marcata differenza nelle predizioni tra CodeBERT e RandomForest con embeddings, così come tra RandomForest e RandomForest senza embeddings

4.2.2 Risultati per Large Class

Come accennato in precedenza, l'identificazione delle Large Class risulta particolarmente complessa per questi classificatori. Analizzando attentamente la distribuzione delle metriche "MCC" e "F1" (mostrate nelle Figure 4.5 e 4.11), emerge chiaramente che le prestazioni dei modelli sono estremamente basse in presenza di questo tipo di smell.

Tabella 4.4: p-value relativi a LC dopo aver effettuato la correzione di Bonferroni su McNemar

Project Name	Smell	RF vs. W2V	RF vs. CB	W2V vs. CB
cassandra-cassandra-1.0.0	LC	6.00	6.00	6.00
hive-release-0.9.0	LC	6.00	6.00	6.00
hsqldb-2.2.8	LC	6.00	1.50	1.50

Per approfondire ulteriormente l'analisi, si è scelto di porre maggiore attenzione a questo smell in particolare, valutando la significatività delle differenze tramite l'utilizzo dei p-value associati alle classi che presentano il smell di "Large Class". Considerando tutti i progetti in cui sono stati individuati code smell di tipo "Large Class", sono stati calcolati i p-value dopo aver applicato il test di Friedman (4.1, 4.2).

Una volta fatto ciò, si è applicato il test di McNemar solo per le istanze che presentavano differenze significative nelle predizioni. I valori del test di McNemar, dopo aver effettuato la correzione di Bonferroni, sono riportati nella tabella: 4.4.

Dai dati emerge che non vi sono differenze significative nelle classificazioni dei code smell, poiché tutti e tre i classificatori mostrano notevole difficoltà nell'identificazione di tali code smell, come illustrato nei grafici 4.2 e 4.8.

4.2.3 Risultati per Complex Class

Per quanto riguarda lo smell in questione, l'analisi delle distribuzione della metrica "MCC" 4.6 evidenzia che il modello di RandomForest raggiunge prestazioni che

variano tra lo 0% e il 25 %. Tuttavia, il modello basato su NLP (CodeBERT) e la RandomForest con embeddings presentano forti difficoltà nel classificare correttamente questa classe, probabilmente a causa di alcune limitazioni intrinseche dei modelli stessi, come la lunghezza dell'input.

Tabella 4.5: p-value relativi a CC dopo aver effettuato la correzione di Bonferroni su McNemar

Project Name	Smell	RF vs. W2V	RF vs. CB	W2V vs. CB
xerces2-j-Xerces-J$_{230}$	CC	3.750000e-01	2.718750	3.00

Inoltre, i risultati del test di Friedman (4.1, 4.2) confermano che, per questo specifico code smell, vi sono differenze significative nelle predizioni dei modelli solo in un caso specifico. Dopo aver applicato il test di McNemar, le differenze più significative tra le predizioni dei modelli si sono verificate tra il modello RandomForest tradizionale e quello con embeddings (4.5).

È importante sottolineare che, nonostante l'assenza di differenze significative nella maggior parte dei casi, le prestazioni generali di tutti e tre i modelli rimangono basse. Questa osservazione suggerisce che per lo smell "Large Class", l'identificazione e la previsione di tale code smell possono essere una sfida complessa, e i modelli attuali potrebbero non essere completamente adeguati per affrontare questa specifica problematica.

4.2.4 Risultati per Refused Bequest

Il grafico 4.4 presenta i risultati dei tre modelli nella classificazione di tale smell, evidenziando che tutti e tre i classificatori riescono difficilmente ad identificare correttamente questo smell con probabilità elevate. La distribuzione delle metriche "MCC" per CodeBERT mostra che questo classificatore riesce a classificare correttamente tale smell con una probabilità che varia tra lo 0% e il 15%, mentre la RandomForest con embeddings e RandomForest tradizionale raggiungono risultati molto bassi, precisamente poco più dello 0%.

Analizzando attentamente i grafici relativi alla distribuzione della metrica "F1" (Figura 4.10), possiamo osservare che tutti e tre i modelli hanno una soglia di partenza minima del 50%. Ciò significa che inizialmente, tutti e tre i modelli hanno una capacità di classificazione che è almeno paragonabile al caso in cui le predizioni vengano fatte in maniera casuale.

Tabella 4.6: p-value relativi a RB dopo aver effettuato la correzione di Bonferroni su McNemar

Project Name	Smell	RF vs. W2V	RF vs. CB	W2V vs. CB
hsqldb-2.2.8	RB	6.000000	6.000000	6.000000
pig-release-0.8.0	RB	3.000000	1.500000	6.000000

L'analisi dei risultati del test di Friedman (4.1, 4.2) indica che solo in due progetti le predizioni dei tre classificatori mostrano differenze significative tra di loro per questo specifico smell. Tuttavia, approfondendo ulteriormente l'analisi utilizzando il test di McNemar, è stato osservato che tali differenze non sono così critiche (4.6). Ciò suggerisce che, nonostante le piccole discrepanze, i tre classificatori hanno prestazioni comparabili nella classificazione di questo particolare smell.

4.2.5 Risultati per Lazy Class

Per quanto riguarda questo specifico tipo di code smell, l'analisi delle distribuzioni delle metriche 4.3 e 4.9 suggerisce che CodeBERT è più efficace rispetto agli altri due classificatori (RandomForest tradizionale e con embeddings). Tuttavia, è importante notare che l'identificazione di questo code smell risulta notevolmente complessa per tutti e tre i classificatori.

A differenza degli altri smell, il test di Friedman 4.1, 4.2 non ha riportato differenze significative tra i gruppi considerati. Di conseguenza, non sono stati ritenuti necessari ulteriori test statistici per confermare o approfondire tali risultati.

4.2.6 Risultati per Spaghetti Code

L'identificazione dello Spaghetti Code come smell risulta particolarmente complessa per questi classificatori. Analizzando attentamente la distribuzione delle me-

triche "MCC" e "F1" (mostrate nelle Figure 4.5 e 4.11), emerge chiaramente che le prestazioni dei modelli sono estremamente basse in presenza di questo tipo di smell. Anche in questo caso il modello che presenta prestazioni leggermente migliori rispetto ad altri modelli, è CodeBERT.

La bassa performance dei modelli potrebbe essere attribuita alla natura intricata e disorganizzata del codice sorgente affetto dallo Spaghetti Code. Questo genere di struttura rende difficile per i classificatori apprendere pattern significativi e stabilire relazioni coerenti tra le variabili coinvolte.

Tabella 4.7: p-value relativi a SC dopo aver effettuato la correzione di Bonferroni su McNemar

Project Name	Smell	RF vs. W2V	RF vs. CB	W2V vs. CB
ant-rel-1.8.3	SC	0.093750	4.359375	0.375000
cassandra-cassandra-1.0.0	SC	6.000000	6.000000	6.000000
hsqldb-2.2.8	SC	1.500000	1.312500	0.093750
lucene-releases-lucene-solr-3.6.0	SC	6.000000	1.500000	0.750000
xerces2-j-Xerces-J$_{230}$	SC	6.000000	6.000000	6.000000

Come per gli altri smell precedentemente analizzati, anche per questo caso sono stati applicati due test statistici per valutare la significatività delle differenze nelle predizioni dei modelli. I risultati dei test mostrano che in tre progetti, le predizioni dei classificatori presentano differenze significative, come evidenziato nelle Tabelle 4.1, 4.2.

È interessante notare che il test di McNemar, utilizzato per valutare specificamente le differenze tra le predizioni dei modelli, non ha identificato differenze significative, come mostrato nella Tabella 4.7. Questo potrebbe suggerire che, nonostante le differenze significative individuate dal test di Friedman, i modelli non sono in grado di distinguere efficacemente tra i diversi progetti affetti da Spaghetti Code.

4.2.7 Minacce alla validità

Come evidenziato in diversi studi che hanno utilizzato lo stesso dataset [2, 3], è importante considerare alcune possibili minacce alla validità del nostro studio.

Una prima possibile minaccia potrebbe essere rappresentata dal dataset utilizzato per la nostra analisi empirica. La selezione del dataset si è basata su diverse considerazioni, come l'eterogeneità dei dati e la presenza di una validazione manuale. Tuttavia, è importante riconoscere che il dataset potrebbe presentare alcune discrepanze o imprecisioni, come errori di etichettatura o la possibilità che alcune istanze positive siano state trascurate. È quindi necessario considerare attentamente queste possibili limitazioni durante l'interpretazione dei risultati [2, 3].

Inoltre, è stata adottata una versione ridotta del dataset a causa delle notevoli risorse computazionali richieste dal modello di NLP, come spiegato nel capitolo due del nostro studio. Tuttavia, poiché il dataset originale era altamente sbilanciato, è stata effettuata una selezione delle classi annotate con un code smell, evitando così la perdita di informazioni rilevanti. Questa scelta è stata guidata dalla necessità di bilanciare le risorse computazionali disponibili con la rappresentatività dei dati.

Un'altra potenziale minaccia riguarda i modelli di apprendimento automatico adottati nel nostro studio. La scelta dei modelli è stata basata su considerazioni specifiche e ricerche precedenti. Ad esempio, il modello di Machine Learning tradizionale è stato selezionato perché in studi precedenti [2, 3] ha dimostrato di avere prestazioni elevate rispetto ad altri modelli di machine learning tradizionae. D'altra parte, l'adozione dei modelli che utilizzano embeddings è stata motivata da ricerche recenti [9] che hanno mostrato risultati promettenti. Infine, il modello di NLP (CodeBERT) è stato scelto in quanto è stato pre-addestrato sul codice sorgente [11, 12] e i modelli BERT-based si sono dimostrati utili per problemi simili [9].

In conclusione, durante questo studio sono state considerate diverse minacce alla validità, come il dataset utilizzato e i modelli di apprendimento automatico adottati. Sono state considerate anche le possibili discrepanze o imprecisioni nel dataset, adottando una versione ridotta per bilanciare le risorse computazionali e assicurandoci che i modelli selezionati fossero supportati da ricerche precedenti.

CAPITOLO 5

Conclusioni e sviluppi futuri

5.1 Conclusioni

La manutenzione e l'evoluzione dei software sono attività complesse e cruciali nel ciclo di vita di un'applicazione. È inevitabile apportare modifiche al codice sorgente per adattarsi a nuovi requisiti, risolvere errori e migliorare le prestazioni. Queste azioni di adeguamento e miglioramento sono fondamentali per mantenere il software rilevante, funzionale e allineato alle esigenze in continua evoluzione degli utenti e del contesto operativo.

Nel corso di questa ricerca, sono stati esaminati tre approcci differenti per l'identificazione di code smell, ognuno con caratteristiche distintive. Il primo metodo si basava sull'uso di machine learning tradizionale, mentre il secondo utilizzava modelli di machine learning tradizionale con vettorizzazioni del codice sorgente come input. Infine, il terzo approccio si fondava sull'utilizzo di modelli di elaborazione del linguaggio naturale (NLP).

L'analisi ha coinvolto un campione di 15 progetti, i cui dettagli sono riportati nella tabella dei progetti (3.1), e sono stati esaminati sei tipi specifici di code smell (tabella dei code smells, 3.2).

Negli ultimi anni, i modelli di elaborazione del linguaggio naturale (NLP) hanno registrato notevoli progressi e suscitato un crescente interesse all'interno della comunità di ricerca. In linea con questo fervente sviluppo, il presente lavoro di tesi si propone di investigare se le tecniche di NLP rappresentino un miglioramento significativo nell'identificazione dei code smell rispetto alle metodologie basate su Machine Learning tradizionale.

I risultati ottenuti da questo studio mostrano che l'identificazione di "code smell" tramite l'uso di modelli di NLP ha mostrato un miglioramento rispetto agli altri modelli utilizzati, sebbene sia ancora lontana da una possibile applicazione pratica. Ciò suggerisce che continuare a investigare sull'uso di questa tecnologia è promettente. Tuttavia, vanno considerati alcuni fattori che potrebbero aver influito sulle prestazioni del modello di NLP. Uno di questi è la scelta del modello stesso, che è stato selezionato tenendo conto di vari trade-off. Inoltre, l'utilizzo del dataset potrebbe aver avuto un impatto poiché inizialmente presentava uno sbilanciamento significativo, e successivamente è stato ridotto ulteriormente per addestrare il modello di NLP. Questa riduzione potrebbe aver influito sulle prestazioni del modello nell'identificazione dei "code smell". Pertanto, ulteriori ricerche e ottimizzazioni possono essere utili per migliorare l'efficacia del modello di NLP in questo contesto.

5.2 Sviluppi futuri

Come evidenziato precedentemente, alcuni vincoli, come le risorse computazionali limitate, hanno influenzato la scelta del modello basato su NLP, risultando quasi una scelta obbligata, a discapito della precisione dei risultati.

Tuttavia, se fossero disponibili risorse computazionali sufficienti, si potrebbe valutare l'adozione di modelli di NLP più potenti e potenzialmente più performanti, come il LongFormer, un modello pre-addestrato in grado di elaborare grandi quantità di dati, riducendo così la perdita di informazioni. Naturalmente, l'utilizzo di tali modelli dovrebbe tenere conto anche del tempo di addestramento necessario, poiché la loro complessità potrebbe richiedere risorse significative.

In ogni caso, è fondamentale bilanciare la complessità del modello con le risorse computazionali e il tempo disponibile, cercando di trovare la migliore combinazione

per ottenere risultati accurati e tempi di elaborazione accettabili. La ricerca di un equilibrio tra prestazioni, precisione e risorse disponibili rappresenta una sfida importante nell'applicazione di tecniche avanzate di NLP per l'identificazione di code smell.

Bibliografia

[1] M. Fowler, "Refactoring: improving the design of existing code, ddisonwesley, 1999." (Citato a pagina 2)

[2] D. S. Manuel, P. Fabio, P. Fabiano, and D. L. Andrea, "Comparing within- and cross-project machine learning algorithms for code smell detection," 2021. [Online]. Available: https://mdestefano.github.io/files/W2.pdf (Citato alle pagine 4, 5, 6, 15, 18, 23, 32 e 51)

[3] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," pp. 1–4, 2019. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8813271?casa_token=U5GGXpb5s-oAAAAA:V-Li68QlXXHNb-3av75nsyV9s9a7eR_vUdwz4zs893VdWPQA3Q5bdNu39zsfMeOpoG-HBo4M (Citato alle pagine 4, 5, 6, 7, 9, 10, 11, 12, 15, 18, 23, 28 e 51)

[4] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," 2013. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6671299?casa_token=MpiJAnrVeZUAAAAA:xdYaNz-yee_h-VNh5EcmTjNjspVV7TtCaZmbjd59dzWUMgThEwLzpQBTBhLsJ2opbbKcmBzB (Citato alle pagine 5 e 12)

[5] H. Liu, Q. Liu, Z. Niu, and Y. Liu, "Automatic metric thresholds derivation for code smell detection," 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7337457?casa_token=dUaj-27oIUQAAAAA:NXmPd8bTXhgtG4QUkviiAsxVC_lnX3pPDBjzgVK-1VxhdrawKXRhsyyOP7jvPDYKciPzPvXz (Citato alle pagine 5 e 6)

[6] F. ArcelliFontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7181590?casa_token=-nXnt2F22CAAAAAA:h0lJpEMfLo07sK31B4H_tJQx3zus3eifvf8rw90ufZwy_VCgi5t1aSB4OKzNBDtG3C_4Bicl (Citato a pagina 5)

[7] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8330265 (Citato a pagina 6)

[8] "Performance and scalability." [Online]. Available: https://huggingface.co/docs/transformers/performance (Citato a pagina 6)

[9] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić, "Automatic detection of long method and god class code smells through neural source code embeddings," pp. 1–3, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422009186 (Citato alle pagine 10, 14, 15, 16, 18, 32 e 51)

[10] E. Cambria and B. White, "Jumping nlp curves: A review of natural language processing research," 2014. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6786458?casa_token=T36gHjufMvwAAAAA:yomxwnCrm2brMP6y-4I--rkJyceTeBmJdQd71_KwQAqMClizIS7e0py5P7DeCGAnP2z6XkmN (Citato a pagina 13)

[11] "Bert." [Online]. Available: https://arxiv.org/abs/1810.04805 (Citato alle pagine 13, 15, 33, 35 e 51)

[12] "Codebert." [Online]. Available: https://huggingface.co/microsoft/codebert-base (Citato alle pagine 13, 33, 35 e 51)

[13] "Cosa sono i transformer." [Online]. Available: https://smartstrategy.eu/intelligenza-artificiale/cosa-sono-i-transformer-e-come-vengono-utilizzati-nellelaborazione-del-linguaggio-natur (Citato a pagina 13)

[14] C. Weiss, T. M.Khoshgoftaar, and D. Wang, "A survey of transfer learning," 2016. [Online]. Available: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-016-0043-6 (Citato alle pagine 13 e 14)

[15] J. Henderson. [Online]. Available: https://aclanthology.org/P05-1023.pdf (Citato a pagina 14)

[16] J. A. L. Marques and S. J. Fong, "Transfer learning," 2022. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/transfer-learning (Citato a pagina 14)

[17] T. S. Himesh Nandani, Mootez Saad, "Dacos-a manually annotated dataset of code smells," 2023. [Online]. Available: https://arxiv.org/abs/2303.08729 (Citato a pagina 23)

[18] "Word2vec." [Online]. Available: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa (Citato alle pagine 32 e 35)

[19] L. Shi, A. Somnali, and P. Guan. (Citato a pagina 35)

[20] "Freidman test." [Online]. Available: https://machinelearningmastery.com/nonparametric-statistical-significance-tests-in-python/ (Citato alle pagine 36 e 39)

[21] "Mcnemar test." [Online]. Available: https://machinelearningmastery.com/mcnemars-test-for-machine-learning/ (Citato a pagina 39)

[22] "Bonferroni." [Online]. Available: https://www.statsimprove.com/it/2020/03/13/correzione-di-bonferroni-come-quando-perche/ (Citato a pagina 39)

Ringraziamenti

Giunto ormai al termine di questo lungo percorso è indispensabile soffermarsi su tutte quelle persone che mi sono state accanto e che, ognuna a modo proprio, mi ha arricchito.

In primis vorrei ringraziare tutta la mia famiglia che mi è sempre stata accanto. Ringrazio mia madre che durante questo percorso mi ha sempre caricato prima di ogni esame, guidato nei momenti più bui e confortato nei momenti più infelici. Ringrazio mio padre che mi ha trasmesso quel pizzico di tranquillità poco prima di sostenere gli esami e che spesso ha ascoltato molte delle mie idee, anche se non sono sicuro che gli sia ben chiaro cosa si faccia ad informatica (la stessa cosa vale per mamma). Infine, ringrazio mia sorella che è stata, e sarà, sempre vittima del mio vagabondare per casa, delle mie sciocchezze e dei miei versi strani fatti prima di ogni esame.

Un ruolo fondamentale in questo percorso è stato ricoperto dalle mie zie. Ringrazio: zia Rosa, zia Marilina e zia Titti. Sono state proprio come delle mamme per me, sempre pronte a darmi supporto, consigli e a spronarmi a dare il massimo. Vi voglio un mondo di bene!

Ringrazio Zio Peppe e Zio Dadi, che forse, un po' come papà, ancora devono capire cosa si faccia ad informatica, però il loro tifo non manca mai.

Ringrazio i miei cugini: Francesca, Fatima, Giuseppe, Laura e Letizia per il loro bene incondizionato e per il forte sostegno durante questo lungo percorso. Devo

ringraziare anche Gianmarco e Nico, i due cugini "aggiunti", che ormai fanno parte di questa famiglia, anche loro hanno visto le mie stranezze e mi hanno accompagnato in questo lungo percorso.

Un sentito ringraziamento va a Marta, che ha sopportato pazientemente le mie innumerevoli domande prima di ogni esame e che dovrà continuare a farlo.

Un ringraziamento speciale va fatto al professore De Lucia che è stato una guida durante questo ultimo anno grazie ai suoi insegnamenti e consigli. Ringrazio il Dottore Manuel De Stefano che ha ascoltato i miei dubbi e mi ha aiutato con ottime indicazioni e suggerimenti.

Ringrazio tutti i ragazzi conosciuti all'università, con i quali ho costruito un rapporto che va ben oltre a quello universitario. Ringrazio Angelo, Francesco, Nicola, Simone, Savino, Valerio e Vito, amici con i quali ho condiviso ore di studio, momenti di allegria e momenti di puro panico e terrore.

E alla fine, voglio proprio ringraziare i miei amici di una vita: Marco Z, Mauro, Federica, Marco P, Clea, Myriam e Giuseppe. Abbiamo iniziato insieme il percorso universitario, ciascuno seguendo la propria passione, ma siamo rimasti sempre uniti nei momenti più difficili e, ovviamente, anche in quelli più leggeri.

Ringrazio, infine tutti coloro che hanno dedicato del tempo per leggere questa tesi. Grazie infinitamente a tutti.