UNIVERSITÀ DI FIRENZE

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

Laboratorio di Data Mining, Code Recommendations

Davide Del Bimbo

ANNO ACCADEMICO 2022/2023

Indice

Capitolo 1

Introduzione

L'analisi relativa al *Code Example Recommendation* è un'attività utile ad assistere gli sviluppatori durante il processo di sviluppo del software. Spesso, infatti, gli sviluppatori dedicano una notevole quantità di tempo nella ricerca di esempi di codice rilevanti su Internet, consultando progetti open-source e documentazione informale. In questo contesto, la documentazione informale, come le discussioni e i forum presenti su Stack Overflow, riveste un ruolo cruciale [1].
A tale scopo, si vuole realizzare un sistema capace di fornire automaticamente degli esempi di codice pertinenti in risposta alle richieste degli sviluppatori.

1.1 Stack Overflow

Stack Overflow rappresenta la più grande e affidabile comunità online dedicata agli sviluppatori che vogliono imparare, condividere le proprie conoscenze di programmazione e costruire la propria carriera [2].
Questo sito Web, appartenente alla rete *Stack Exchange*, offre ai programmatori la possibilità di porre domande relative ad argomenti informatici e di ricevere risposte dagli altri membri della comunità.
Tale piattaforma funge da punto d'incontro per gli utenti, permettendo loro di presentare le proprie domande e contribuire con delle risposte. Inoltre, il sistema di votazione consente di esprimere apprezzamenti sia per le domande che per le risposte, contribuendo così a valutarne la qualità e l'utilità. Questo sistema di votazione aiuta a distinguere le risposte migliori dalle meno utili, garantendo che le risorse di alta qualità siano facilmente accessibili a tutti gli sviluppatori.
Stack Overflow offre una vasta gamma di argomenti di programmazione, coprendo numerosi linguaggi di programmazione, framework, librerie e strumenti. Gli utenti possono cercare domande e risposte esistenti per trovare soluzioni ai loro problemi, oppure possono porre nuove domande se non trovano una risposta adeguata.
Perciò, vogliamo realizzare un sistema di Code Example Recommendations basato sui post di Stack Overflow. Questa scelta ci consente di creare un dataset coerente e ricco di informazioni, fondamentale per il nostro obiettivo.

1.2 Code Example Recommendations

La **Code Example Recommendations** è una disciplina che si concentra nel fornire suggerimenti e soluzioni di codice pertinenti agli sviluppatori. Questa pratica si basa sull'idea di rendere più efficiente e produttiva l'esperienza di sviluppo software.
L'obiettivo principale della Code Example Recommendations è quello di ridurre il tempo e lo sforzo necessari per individuare soluzioni a problemi di programmazione, utilizzando tecniche di apprendimento automatico per analizzare grandi quantità di codice sorgente e identificare possibili soluzioni. Quando uno sviluppatore cerca di risolvere un problema specifico, il sistema di Code Example Recommendations può suggerire automaticamente frammenti di codice pertinenti da utilizzare come punto di partenza o come ispirazione.

1.3 Locality Sensitive Hashing

La **Locality Sensitive Hashing** (LSH) è una tecnica utilizzata nel campo del Data Mining ed, in particolare, dell'Information Retrieval per identificare oggetti simili in grandi dataset. L'obiettivo principale di LSH è quello di ridurre la complessità computazionale associata alla ricerca di vicini più simili in grandi spazi multidimensionali. Questo è particolarmente utile quando si devono confrontare oggetti complessi come documenti o vettori di features.
LSH opera mediante l'uso di funzioni hash che mappano gli oggetti in uno spazio hash. Queste funzioni sono progettate in modo che oggetti simili abbiano una maggiore probabilità di essere mappati nello stesso bucket o nella stessa regione hash. Pertanto, quando si cerca un oggetto simile, invece di confrontarlo con tutti gli elementi del dataset, è sufficiente cercare solo all'interno del bucket o della regione hash corrispondente, riducendo notevolmente il tempo di ricerca.

1.4 BERT

Le **Neural Networks**, ispirate dalla struttura e dal funzionamento dei neuroni del cervello, rappresentano un potente modello di intelligenza artificiale ampiamente utilizzato nel campo del Machine Learning, in particolare quando la relazione tra input e output è complessa e non lineare. Tipicamente, le Neural Networks sono impiegate per apprendere lunghe sequenze di testo, al fine di estrarre informazioni semantiche dai token e dalle frasi. Questo approccio permette di effettuare previsioni sulle parole successive in una sequenza e di comprendere il contesto di una conversazione.
Tuttavia, le Neural Networks presentano due problemi principali. Il primo problema riguarda la lentezza nell'addestramento, poiché tali reti processano i token in modo sequenziale, richiedendo un notevole tempo per l'addestramento su lunghi testi. Il secondo problema è legato alla gestione di lunghe sequenze di testo

a causa dell'effetto di vanishing. Questo fenomeno si verifica durante il processo di addestramento, quando vengono applicati metodi basati sul gradiente (come la backpropagation), e implica l'impossibilità di addestrare correttamente i layer iniziali della Neural Network. Questo può comportare un aumento dei tassi di errore e una minore accuratezza nei compiti di analisi del testo.

I modelli **Transformers** sono una nuova architettura progettata per risolvere i problemi precedentemente menzionati nelle Neural Netwroks. Un modello Transformer è composto da due componenti principali: l'Encoder e il Decoder. L'Encoder estrae features rilevanti dai dati di input, mentre il Decoder utilizza queste features estratte per generare risultati (come la traduzione di un linguaggio in un altro o la risposta a una domanda). Questo sistema si basa su tre concetti:

- **Codifica posizionale**: consente al modello di mantenere informazioni sull'ordine delle parole all'interno di una frase.
- **Attention**: consente di assegnare un peso maggiore alle features più rilevanti dei dati (ad esempio, all'interno di una frase, il soggetto potrebbe essere più importante di altri elementi).
- **Self-Attention**: facilita la comprensione di una parola in base al contesto fornito dalle parole circostanti (ad esempio, la parola "conto" può indicare sia l'azione di contare un insieme di oggetti che un deposito bancario).

Il **Bidirectional Encoder Representations from Transformers** (**BERT**) è una famiglia di modelli linguistici introdotta nel 2018 dai ricercatori di Google. BERT è un modello basato su Transformer, utilizzato per la rappresentazione del linguaggio.
L'algoritmo BERT segue un processo diviso in due fasi principali: il pre-training e il fine-tuning. Nella fase di pre-training, BERT viene addestrato su un vasto corpus di testi, utilizzando il Toronto BookCorpus (800 milioni di parole) e la versione in inglese di Wikipedia (2.500 milioni di parole). In questa fase, i testi vengono selezionati dai documenti per estrarre la semantica in modo migliore. Nella fase di fine-tuning, BERT viene applicato ad un dataset o a un compito specifico, ma i suoi parametri iniziali sono già stati pre-addestrati durante la fase precedente. In altre parole, il modello pre-addestrato viene personalizzato per adattarsi ai nuovi dati o compiti.
Poiché l'obiettivo di BERT è quello di generare un modello di rappresentazione del linguaggio, necessita solo della parte di Encoder. Questo modello sfrutta la statistica per scoprire le relazioni tra le parole e applica la codifica posizionale, l'Attention e la Self-Attention per incorporare le informazioni semantiche in modo efficace. In particolare, l'Attention assegna maggiore peso ai token all'interno di una frase in base alla loro rilevanza, mentre la Self-Attention viene applicata in modo bidirezionale per consentire di catturare informazioni sia dai lati sinistri che da quelli destri dei token. In figura 1.1 viene illustrata l'architettura di un modello $\text{BERT}_{\text{BASE}}$, costituito da 12 Encoder.

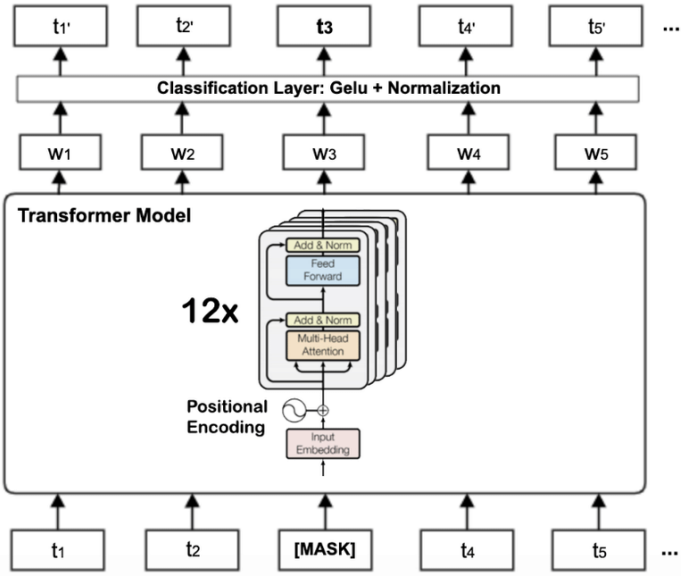


Figura 1.1: BERT Embedding.

L'input dell'Encoder di BERT consiste in una sequenza di token, i quali vengono prima convertiti in vettori e quindi elaborati nella rete. Prima che l'elaborazione possa iniziare, BERT richiede che l'input venga sottoposto a un processo di embedding attraverso tre livelli distinti. La figura 1.2 illustra questo processo di embedding, dimostrando come BERT utilizzi questi tre livelli per catturare i contesti dei testi:

- **Token embedding**: codifica ogni token e parola in valori numerici. Di particolare importanza sono due token speciali: `[CLS]`, aggiunto all'inizio del testo, e `[SEP]`, utilizzato per separare le frasi.
- **Sentence embedding**: attribuisce un'etichetta ai token in base alla frase di appartenenza.
- **Positional embedding**: aggiunge informazioni sulla posizione dei token nella sequenza.

Il principale vantaggio di BERT rispetto ad altri modelli, come Word2Vec, sta nel fatto che, oltre a catturare una rappresentazione statica delle parole, incorpora anche la rappresentazione delle parole circostanti. Ciò significa che una parola può avere rappresentazioni diverse in contesti differenti a causa dell'influenza delle parole circostanti.

L'approccio BERT presenta notevoli vantaggi rispetto ai modelli basati su Neural Networks tradizionali, come le *Recurrent Neural Networks* (RNN) e le *Long Short-Term Memory* (LSTM), che leggono il testo in modo sequenziale (cioè da sinistra a destra). In contrasto, BERT opera sulla base delle frasi anziché delle singole parole, consentendo di estrarre il significato delle parole nel contesto più

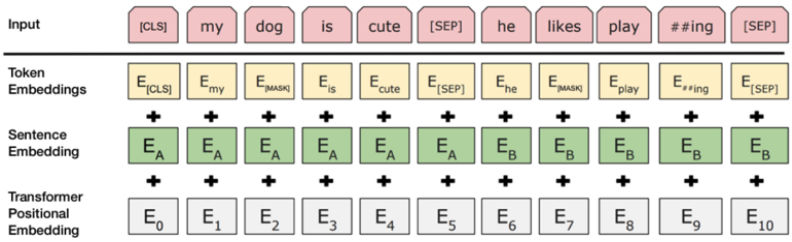


Figura 1.2: BERT Embedding.

ampio. Questa capacità di comprendere un'intera frase aiuta a catturare il significato delle parole in base al contesto, portando a risultati più accurati in diversi compiti (come la risposta a domande, la previsione della frase successiva, l'inferenza semantica ecc...).

Capitolo 2

Metodologia

La ricerca si rivolge in particolare agli esempi di codice Java presenti su Stack Overflow, in risposta a domande pubblicate entro dicembre 2022.
Una volta effettuata una fase di pre-elaborazione dei dati, gli esempi di codice vengono tradotti in vettori numerici (ovvero *embeddings*) mediante l'utilizzo di un modello BERT base `bert-base-uncased` e di un modello sottoposto a fine-tuning. Successivamente, si applica un algoritmo basato su Local Sensitive Hashing (LSH) noto come Query-Aware LSH. Questo algoritmo è progettato per ottimizzare lo spazio di ricerca e individuare gli esempi di codice che presentano somiglianze con la query formulata dall'utente (fig 2.1).

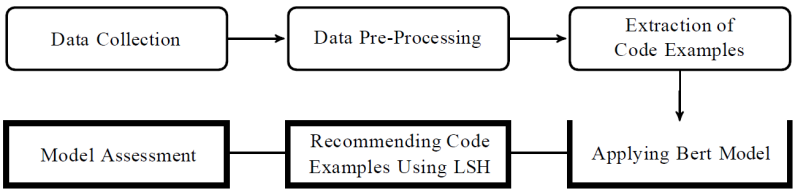


Figura 2.1: Fasi principali della metodologia seguita.

2.1 Collezione dei dati

Il dataset è costituito da esempi di codice estratti dai post di Stack Overflow e conservati in file dump, accessibili pubblicamente sul database di Stack Exchange. In particolare, tali file dump comprendono tutte le discussioni pubblicate su Stack Overflow dal 2008 fino al 2022.
Il dataset fornito da Stack Exchange è costituito da entità che rappresentano post di domanda e di risposta. Ogni entità contiene diversi attributi che mostrano vari aspetti di un post. In particolare, gli attributi necessari all'analisi sono [3]:

- *Id*: identificativo univoco del post;
- *PostType*: indica se il post corrisponde ad una domanda (`PostType=1`) o ad una risposta (`PostType=2`);

- *AcceptedAnswerId* (presente solo se `PostType=1`): identificativo del post contrassegnato come risposta ad una domanda;
- *ParentId* (presente solo se `PostType=2`): identificativo del post domanda relativo alla risposta;
- *Body*: contenuto del post (in formato HTML);
- *Tag*: etichette assegnate al post (come i linguaggi di programmazione a cui fa riferimento);
- *Score*: punteggio assegnato al post dalla comunità.

2.2 Pre-Processing dei dati

Una volta acquisito il dataset, è stato eseguito un processo di pre-elaborazione dei dati articolato in quattro fasi, implementate mediante la classe `DataProcessor`:

1. Poiché siamo interessati esclusivamente agli esempi di codice relativi al linguaggio di programmazione Java, è stata applicata una strategia di filtraggio. In particolare, sono stati considerati solamente i post che includevano il tag `<Java>`. Contemporaneamente, sono stati esclusi tutti i post che includevano anche esempi di codice in altri linguaggi di programmazione (come Javascript e C).
2. Sono stati recuperati tutti i post il cui *Id* corrispondeva all'*AcceptedAnswerId*. Tuttavia, è importante notare che alcune domande su Stack Overflow potrebbero non avere risposte, con il risultato di avere un campo *AcceptedAnswerId* vuoto.
3. Sono stati esclusi tutti i post con un punteggio (*Score*) inferiore ad una soglia prefissata, impostata a `threshold=2`.
4. Infine, è stato estratto il testo contenuto nel campo *Body*, relativo alle risposte ottenute dalle precedenti fasi di filtraggio.

2.3 Estrazione degli esempi di codice

Dopo aver estratto i post di risposta interessati, è stato necessario recuperare gli esempi di codice a causa della loro integrazione con i commenti in linguaggio naturale. In particolare, gli esempi di codice sono identificabili nel testo dai tag HTML `<pre><code>` e `</code></pre>`.
Tuttavia, al fine di garantire la qualità degli esempi di codice, sono stati esclusi quelli con una lunghezza inferiore a 100 caratteri e quelli che iniziano con determinati caratteri speciali (come "$", "/" o "@").

Al termine di questo processo di filtraggio, in cui sono stati elaborati 57.721.549 post, è stato ottenuto un dataset contenente 178.610 esempi di codice in linguaggio Java.

2.4 Applicazione del modello BERT

Per applicare il modello BERT agli esempi di codice estratti, sono stati utilizzati i Transformer del modulo Python `sentence-transformers`. In particolare, è stato impiegato il modello BERT pre-addestrato `bert-base-uncased`. Tale modello mappa frasi e paragrafi in uno spazio vettoriale denso di 768 dimensioni ed è stato addestrato su tutti i dati di addestramento disponibili (oltre 1 miliardo di coppie di addestramento) [4]. Perciò, ogni campione di esempio di codice è stato inserito in input al modello BERT, che incorpora le informazioni basate sui token e sulla semantica in un vettore di dimensioni (1×768).

2.4.1 Fine-Tuning

Il processo di fine-tuning è iniziato utilizzando un modello pre-addestrato che aveva già appreso rappresentazioni linguistiche generiche da un vasto corpus di testi. Successivamente, il modello è stato ulteriormente addestrato in due fasi sequenziali, utilizzando il dataset di esempi di codice Java estratto in precedenza:

1. Nella prima fase, per migliorare la capacità del modello di comprendere e rappresentare efficacemente gli esempi di codice Java, è stato eseguito un processo di fine-tuning utilizzando un **Transformer-based Denoising AutoEncoder** (TSDAE). Questa rete neurale è stata progettata per ricostruire il codice sorgente a partire da versioni con rumore o informazioni mancanti. Il TSDAE è stato addestrato in modo non supervisionato utilizzando il dataset di esempi di codice Java non annotato.
   Durante questa fase di addestramento, il TSDAE ha ricevuto versioni del codice con rumore, in cui parti casuali di parole o caratteri sono state rimosse. L'obiettivo della rete è stato quello di codificare gli input danneggiati in vettori di dimensioni fissate e quindi richiedere al decodificatore di ricostruire le frasi originali a partire da queste rappresentazioni vettoriali (fig 2.2).

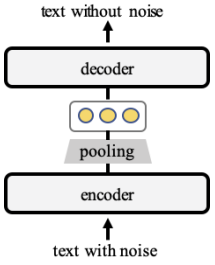


Figura 2.2: Architettura TSDAE.

Al termine di questa fase di fine-tuning, ci aspettiamo che il modello abbia sviluppato una migliore comprensione del linguaggio specifico del codice Java. In questo modo possiamo utilizzare il modello per generare rappresentazioni (embeddings) di codice migliorate [5].

2. Nella seconda fase, per migliorare la capacità del modello di recuperare codici pertinenti in risposta a query specifiche, è stato eseguito un ulteriore processo di fine-tuning utilizzando la **Multiple Negative Ranking Loss** (MNR). Questa funzione di loss è particolarmente adatta per l'addestramento di embeddings in applicazioni di retrieval in cui si dispone soltanto di coppie positive, come ad esempio coppie (query, codice rilevante) [6].
In questo contesto, la loss MNR richiede come input un batch costituito da n coppie di campioni (a_i, p_i), dove ogni coppia è formata da un'ancora a_i e da un positivo p_i. Per ognuna di queste coppie, si assume che le restanti $n-1$ coppie (a_i, p_j), con $j \neq i$, all'interno del batch rappresentino coppie negative (fig 2.3).

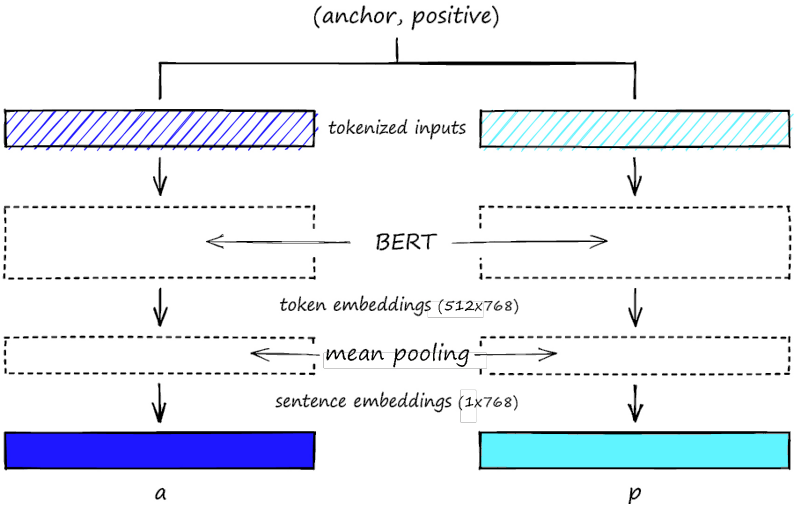


Figura 2.3: La coppia (ancora, positivo) viene elaborata in maniera distinta, con l'impiego di un livello di mean pooling che trasforma gli embeddings dei token in embeddings rappresentativi dei codici.

In pratica, per ciascuna ancora a_i, la loss MNR utilizza il campione p_i come campione positivo e tutti gli altri p_j come campioni negativi. Pertanto, all'interno di un batch, ogni ancora a_i dispone di un campione positivo p_i e di $n-1$ campioni positivi p_j.
L'obiettivo della loss MNR è quello di minimizzare la Negative Log-Likelihood, assicurando che le coppie positive ottengano punteggi più elevati rispetto alle coppie negative. In altre parole, ci aspettiamo che gli aggiornamenti della loss inducano il modello ad aggiornare i suoi pesi in modo che le coppie positive siano più vicine e quelle negative più lontane (fig 2.4).

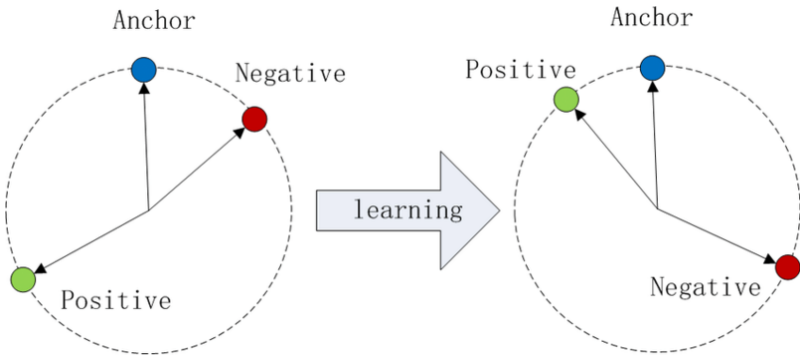


Figura 2.4: Gli esempi positivi vengono avvicinati all'ancora, mentre gli esempi negativi vengono allontanati.

Al termine di questa fase di fine-tuning, ci aspettiamo che il modello abbia sviluppato una migliore capacità di retrieval di codici Java relativi a query specifiche [4].

2.5 Raccomandazione degli esempi di codice attraverso LSH

Dopo aver rappresentato gli esempi di codice in vettori numerici, sono stati considerati due diversi algoritmi LSH per affrontare il task di Code Example Recommendation: Random Hyperplane-based e Query-Aware. Il primo assegna i campioni di dati ai bucket senza tenere conto della query, mentre il secondo prende in considerazione la query durante l'assegnazione dei campioni di dati ai bucket.

2.5.1 Random Hyperplane-based

In questo approccio, scegliamo un vettore casuale $\vec{r}$ di dimensione d dalla distribuzione Gaussiana. In corrispondenza di questo vettore $\vec{r}$, definiamo una funzione hash $h_{\vec{r}}$ come segue:

$$h_{\vec{r}}(\vec{u}) = \begin{cases} 1 & \text{se } \vec{r} \cdot \vec{u} \geq 0 \\ 0 & \text{se } \vec{r} \cdot \vec{u} < 0 \end{cases}$$

Pertanto, la probabilità che due vettori $\vec{u}$ e $\vec{v}$ vengano assegnati allo stesso bucket è data da:

$$\Pr(h_{\vec{r}}(\vec{u}) = h_{\vec{r}}(\vec{v})) = 1 - \frac{\theta(\vec{u}, \vec{v})}{\pi}$$

dove $\theta(\vec{u}, \vec{v})$ rappresenta una misura di similarità nota come *somiglianza coseno*, definita come segue:

$$\theta(\vec{u}, \vec{v}) = \cos^{-1}\left(\frac{|\vec{u} \cap \vec{v}|}{\sqrt{|\vec{u}| \cdot |\vec{v}|}}\right)$$

Questa formula ci indica che la probabilità di assegnare due vettori allo stesso bucket mediante LSH è inversamente proporzionale all'angolo compreso tra i vettori stessi. In altre parole, quanto più sono simili i vettori (ossia quanto più è piccolo l'angolo tra di essi), tanto maggiore è la probabilità che vengano collocati nello stesso bucket. Pertanto, se due campioni di dati sono simili nella loro rappresentazione binaria, allora la probabilità di assegnarli agli stessi valori hash risulta essere più elevata.

Per la raccomandazione di esempi di codice, sfruttiamo l'algoritmo Random Hyperplane che consente di ridurre la dimensionalità dei vettori numerici ed effettuare una comparazione tra i vettori risultanti. In particolare, i vettori ridotti vengono convertiti da rappresentazioni decimali a rappresentazioni binarie e successivamente suddivisi in bucket in base ai loro valori binari. Questo approccio ci consente di individuare elementi simili ad una determinata query in uno spazio di dimensioni ridotte, facendo uso sia dei campioni di dati (cioè gli esempi di codice) sia dei vettori di query.
Nel nostro contesto, seguiamo i seguenti passaggi. Ogni frammento di codice e il testo della query vengono convertiti in vettori numerici di dimensione 1×768. Pertanto, per un insieme di N campioni di dati (ovvero, per N esempi di codice), la dimensione del vettore dati sarà di $N \times 768$. In seguito, applichiamo il seguente algoritmo:

Algorithm 1 LSH basato su Random Hyperplane.

Input: D as Data samples vector, Q as Query vector, R as a Randomized vector and M as the number of hash tables.

for $i \longleftarrow 1 \dots M$ **do**
    # *Data Samples Vector*
    $\text{Result}_{N\times K} \longleftarrow D_{N\times 768} \cdot R_{768\times K}$;
    $\text{SgnResult} \longleftarrow \text{Sign}(\text{Result}_{N\times K})$;
    If the items of the SgnResult are negative assign 0, otherwise they are already 1;
    $\text{BucketId} \longleftarrow \sum_{i=0}^{K-1} 2^i \cdot \text{SgnResult}[i]$
    # *Query Vector*
    $\text{QResult}_{1\times K} \longleftarrow Q_{1\times 768} \cdot R_{768\times K}$;
    $\text{SgnQResult} \longleftarrow \text{Sign}(\text{QResult}_{1\times K})$;
    If the items of the SgnQResult are negative assign 0, otherwise they are already 1;
    $\text{QBucketId} \longleftarrow \sum_{i=0}^{K-1} 2^i \cdot \text{SgnQResult}[i]$
Retrieve the samples (u) that are at the same bucket as query vector (q) from all of the hash tables and rank them based on Cosine similarity;
Retrieve top N similar samples as recommendation items;

In questo algoritmo, il vettore D relativo ai campioni di dati (di dimensione $N \times 768$) viene moltiplicato con il vettore random R precedentemente generato (di dimensione $768 \times K$). Questa procedura consente di ridurre la dimensionalità dei campioni di dati, mappandoli in uno spazio di dimensione inferiore K (ad esempio,

da 768 a K). Il numero di tabelle hash M determina quante volte questo processo viene ripetuto.
I valori risultanti vengono convertiti in rappresentazioni binarie (1 o 0) in base al fatto che siano maggiori o minori di 0. Successivamente, ogni campione di dati viene assegnato ad uno specifico bucket in base ai suoi valori binari. Ogni tabella hash dispone di 2^K bucket e, di conseguenza, ogni campione di dati viene assegnato ad uno di questi bucket in base alle sue rappresentazioni binarie.
Lo stesso procedimento viene applicato al vettore Q relativo alla query, assegnando anch'esso ad uno specifico bucket.
Infine, per ogni tabella hash, vengono raccolti tutti i campioni di dati il cui bucket coincide con quello del vettore della query.
Successivamente, viene calcolata la somiglianza coseno tra ognuno di questi campioni di dati e la query. I risultati vengono ordinati in base ai valori di somiglianza, dall'alto verso il basso. Alla fine, vengono restituite all'utente le prime N raccomandazioni.

2.5.2 Query-Aware

In questo approccio, la query influenza direttamente l'assegnazione dei campioni di dati ai bucket. In questo contesto, i campioni di dati rappresentano gli esempi di codice, mentre la query rappresenta una richiesta in linguaggio naturale oppure un metodo API.
L'approccio Query-Aware risolve il problema dello spostamento casuale che può verificarsi durante la mappatura dei campioni di dati in una dimensione inferiore al fine di assegnarli ai bucket.
Analizziamo l'algoritmo Query-Aware:

Algorithm 2 LSH basato su Query-Aware.

Input: D as Data samples vector, Q as Query vector, R as a Randomized vector, M as the number of hash tables and ℓ as the threshold of the occurrence of a data sample.

1: **for** $i \longleftarrow 1 \dots M$ **do**
2: Impose hash function h_i by multiplying object O_i and query vector q to randomized vector R;
3: Increase #Col(O_i) if the $|h_i(O_i) - h_i(q)| \leq \frac{w}{2}$;
4: **if** #Col(O_i) $\geq \ell$ **then**
5: $C = C \cup O_i$;
6: Calculate the Euclidean distance between O_i in C and q;
7: Sort the Euclidean distances incrementally;
8: Retrieve top N similar samples as recommendation items from the sorted list;

Secondo questo algoritmo, in base al numero di tabelle hash M, vengono generati dei vettori random R. Per ciascuna di queste tabelle hash, il vettore D relativo ai

campioni di dati viene moltiplicato con il vettore random R e vengono generati i relativi valori hash. Quindi, si misura la distanza Euclidea tra il valore hash di ciascun campione di dati ed il valore hash del vettore query. Qualora la distanza Euclidea risulti inferiore a $\frac{w}{2}$, dove w indica la dimensione dei bucket, allora si incrementa il numero di occorrenze di questi campioni. Se il numero di occorrenze di un campione di dati è maggiore o uguale alla soglia ℓ, che rappresenta il minimo numero di volte in cui la distanza Euclidea tra il vettore hash di un campione di dati e il vettore hash della query deve essere minore o uguale a $\frac{w}{2}$, allora il campione di dati viene incluso nell'insieme dei candidati C.
Dopo aver raccolto i campioni candidati, si calcola la loro distanza Euclidea rispetto al vettore query e si procede ad una classificazione crescente in base a queste distanze. Infine, vengono raccomandati i primi N campioni di dati come esempi di codice più simili alla query.
Questo approccio filtra i campioni di dati in base a una determinata query in due fasi. In primo luogo, considera i campioni in base alla somiglianza del loro valore hash con il valore hash della query. Successivamente, i campioni filtrati dalla prima fase vengono selezionati in base alla loro somiglianza con la query.
Nella nostra implementazione, abbiamo impostato il valore di soglia $\ell = 2$, il che significa che se un campione di dati occorre nello stesso bucket del vettore query almeno due volte, questo verrà aggiunto all'insieme dei candidati C.

2.5.3 Valutazione dei modelli

Valutiamo le prestazioni degli algoritmi utilizzando quattro parametri distinti:

- **HitRate**: dato un insieme di query Q, allora *HitRate@k* misura la percentuale di query che hanno restituito almeno un risultato rilevante tra i primi k elementi raccomandati. Possiamo definire questo parametro attraverso questa formula:

$$\text{HitRate@k} = \frac{1}{|Q|} \sum_{q \in Q} H(R(Q), k)$$

  In questa equazione, Q rappresenta un insieme di query, mentre la funzione $H(R(Q), k)$ restituisce 1 se c'è almeno un risultato rilevante tra i primi k elementi raccomandati e 0 in caso contrario. Un valore prossimo a 1 per questo parametro indica che il sistema di raccomandazione è efficace nell'individuare risultati rilevanti.

- **Mean Reciprocal Rank** (**MRR**): tale variabile si calcola come la media inversa del primo rango relativo agli elementi raccomandati quando si verifica l'HitRate. In altre parole, rappresenta la media degli inversi delle posizioni in cui vengono trovati i primi risultati rilevanti per ciascuna query (ad esempio, se abbiamo eseguito due query e i risultati rilevanti si trovano al secondo e al

terzo posto nelle raccomandazioni, l'MRR è la media di $\frac{1}{2} + \frac{1}{3}$). La formula seguente definisce la metrica MRR:

$$\text{MRR} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{First rank of the relevant result}}$$

Un valore elevato di MRR indica che i risultati rilevanti sono spesso posizionati nelle prime posizioni delle raccomandazioni, il che riflette una migliore efficienza del sistema di raccomandazione.

- **Average Execution Time**: valutiamo il tempo di esecuzione di un algoritmo in due fasi. Nella prima fase, misuriamo il tempo necessario per applicare gli algoritmi di hashing ai campioni di dati. Nella seconda fase, misuriamo il tempo richiesto per restituire i risultati quando l'hashing è già stato applicato.
- **Relevance**: questa metrica rappresenta un punteggio qualitativo assegnato dagli sviluppatori agli elementi raccomandati. I punteggi di rilevanza sono assegnati in base alla tabella 2.1:

Score	Descrizione
0	Nessun risultato utile restituito
1	Il risultato restituito è poco rilevante
2	Ci sono alcuni accenni, ma ancora fuori contesto
3	Contiene caratteristiche rilevanti, ma non le principali
4	Il risultato è nel contesto della query ed è utile

Tabella 2.1: Scala di punteggio della metrica di rilevanza.

Per la valutazione degli algoritmi basati su LSH, esaminiamo due categorie di query, ovvero quelle basate sui metodi API e quelle espresse in linguaggio naturale. Queste categorie sono dettagliate nella tabella 2.2.

Query ID	API	Natural Language
Query 1	Jpanel.add()	How to add an image to a JPanel?
Query 2	StringBuilder.append()	How to generate a random alpha-numeric string?
Query 3	Writer.write()	How do I create a file and write to it?
Query 4	Method.invoke()	How do I invoke a Java method when given the method name as a string?
Query 5	Jsoup.parse()	Remove HTML tags from a String?
Query 6	URLDecoder.decode()	How to get the path of a running JAR file?
Query 7	MessageDigest.digest()	Getting a File's MD5 Checksum in Java
Query 8	Properties.load()	Loading a properties file from Java package
Query 9	AudioSystem.getAudioInputStream()	How can I play sound in Java?
Query 10	JSch.getSession()	What is the best way to SFTP a file from a server?
Query 11	BufferedReader.readLine()	How to read a CSV file?
Query 12	MessageDigest.getInstance()	How to generate MD5 hash code?
Query 13	DatagramSocket.send()	How to send a packet via UDP?
Query 14	String.split()	How to split a string?
Query 15	MediaPlayer.play()	How to play an audio file?
Query 16	FTPClient.storeFile()	How to upload a file to FTP?
Query 17	thread.start()	How to initialize a thread?
Query 18	DriverManager.getConnection()	How to connect to a JDBC database?
Query 19	zipFile.getInputStream()	How to read a ZIP archive?
Query 20	MimeMessage.setFrom()	How to send an email?

Tabella 2.2: Query utilizzate come test per confrontare i modelli BERT.

2.5.4 Osservazioni sull'analisi

Dato che l'algoritmo LSH Random Hyperplane-based ha dimostrato prestazioni non soddisfacenti durante le fasi sperimentali, abbiamo deciso di orientare l'analisi sull'algoritmo Query-Aware. In particolare, ci concentriamo sui risultati ottenuti utilizzando i due modelli BERT per mettere in luce l'efficacia del processo di fine-tuning descritto in 2.4.1.

Capitolo 3

Risultati

Analizziamo i risultati ottenuti dalla valutazione empirica condotta per valutare le raccomandazioni di esempi di codice generate dall'approccio Query-Aware LSH proposto, con particolare attenzione al confronto tra il modello BERT base (`bert-base-uncased`) ed il modello BERT sottoposto a fine-tuning secondo la procedura discussa in 2.4.1.

3.1 HitRate

Innanzitutto, analizziamo le prestazioni dei due modelli esaminati in termini di HitRate. In particolare, la HitRate@k misura la frazione di query che hanno restituito almeno un risultato rilevante tra i primi k elementi raccomandati.

	`bert-base-uncased`				**TSDAE + MNR**			
	API		**Natural Language**		**API**		**Natural Language**	
Query ID	**Top 10**	**Top 20**	**Top 10**	**Top 20**	**Top 10**	**Top 20**	**Top 10**	**Top 20**
Query 1	1	1	1	1	1	1	1	1
Query 2	1	1	0	0	1	1	1	1
Query 3	0	0	0	1	1	1	1	1
Query 4	0	0	0	0	1	1	0	1
Query 5	0	1	0	1	1	1	1	1
Query 6	1	1	0	0	1	1	1	1
Query 7	0	1	0	0	1	1	0	0
Query 8	0	1	0	0	1	1	1	1
Query 9	1	1	0	0	1	1	1	1
Query 10	0	0	0	0	0	0	1	1
Query 11	1	1	0	0	1	1	1	1
Query 12	1	1	1	1	1	1	1	1
Query 13	0	0	0	1	1	1	1	1
Query 14	1	1	0	0	1	1	1	1
Query 15	1	1	0	0	1	1	1	1
Query 16	0	0	0	0	1	1	1	1
Query 17	1	1	1	1	1	1	1	1
Query 18	0	0	1	1	1	1	1	1
Query 19	0	1	0	0	1	1	1	1
Query 20	0	0	0	0	1	1	1	1
HitRate	0.45	0.65	0.2	0.35	0.95	0.95	0.9	0.95

Tabella 3.1: Risultati della misura di HitRate ottenuti per i modelli BERT.

La tabella 3.1 mostra i valori di HitRate ottenuti dall'algoritmo Query-Aware LSH, utilizzando gli embeddings generati sia dal modello `bert-base-uncased` che dal modello sottoposto a fine-tuning. Tali risultati sono derivati dall'esecuzione di query formulate sia in formato API che in linguaggio naturale.
Come possiamo osservare dalla tabella 3.1, i valori di HitRate per le query in formato API sono $(0.45, 0.65)$ e $(0.95, 0.95)$ rispettivamente per il modello `bert-base-uncased` ed il modello sottoposto a fine-tuning, considerando le prime 10 e 20 raccomandazioni generate.
Analogamente, i valori di HitRate per le query in linguaggio naturale sono $(0.2, 0.35)$ e $(0.9, 0.95)$ rispettivamente per il modello `bert-base-uncased` ed il modello fine-tuned, considerando sempre le prime 10 e 20 raccomandazioni generate.
Questi risultati evidenziano l'efficacia del modello sottoposto a fine-tuning nel fornire raccomandazioni rilevanti tra le prime k raccomandazioni generate, sia per le query in formato API che in linguaggio naturale. Al contrario, il modello `bert-base-uncased` mostra prestazioni notevolmente inferiori sia per le query in formato API che in linguaggio naturale, con particolare criticità nel caso di query in linguaggio naturale.
In effetti, il modello sottoposto a fine-tuning ha subito un processo di addestramento specifico, focalizzato sull'interpretazione del codice Java e sull'ottimizzazione della corrispondenza tra query e risposte. Questa specializzazione consente al modello di dimostrare una maggiore efficacia nel fornire risultati rilevanti tra le prime k raccomandazioni generate. Di conseguenza, l'addestramento ha contribuito notevolmente all'abilità del modello di rispondere in modo più efficace alle query in formato API ed in linguaggio naturale, evidenziando la sua superiorità rispetto al modello base `bert-base-uncased`.

3.2 Mean Reciprocal Rank

Analizziamo adesso i risultati della misura MRR per i due modelli considerati, basati sui due tipi di query esaminati. In particolare, il Mean Reciprocal Rank (MRR) è calcolato come la media dell'inverso del rango del primo elemento raccomandato, quando si verifica un HitRate, ossia quando viene individuato un risultato rilevante.
La tabella 3.2 presenta i valori di MRR ottenuti rispettivamente per il modello `bert-base-uncased` e per il modello sottoposto a fine-tuning, utilizzando sia query in formato API che in linguaggio naturale. Dai risultati ottenuti emerge chiaramente che il modello sottoposto a fine-tuning dimostra un'elevata efficacia nel fornire raccomandazioni utili tra le prime proposte. Inoltre, è interessante notare che i risultati ottenuti con le query in formato API comportano un MRR più elevato. In particolare, si osserva un valore di MRR pari a $(0.84, 0.71)$ rispettivamente per le query in formato API ed in linguaggio naturale. Questo risultato potrebbe essere attribuito al processo di fine-tuning che consente al

modello di interpretare con precisione i metodi richiesti, offrendo raccomandazioni più pertinenti tra i primi risultati generati.
Al contrario, il modello `bert-base-uncased` mostra risultati inferiori in termini di MRR, registrando valori di $(0.29, 0.07)$ rispettivamente per le query in formato API e in linguaggio naturale. Questi risultati indicano una limitata capacità del modello di fornire raccomandazioni rilevanti nelle prime posizioni. In particolare, il valore MRR particolarmente basso ottenuto per le query in linguaggio naturale potrebbe suggerire che il modello, essendo addestrato su un corpus prevalentemente in linguaggio naturale, potrebbe non essere predisposto a fornire soluzioni in formato di codice Java. La differenza sostanziale nei valori di MRR tra i due modelli evidenzia ulteriormente il beneficio del processo di fine-tuning.

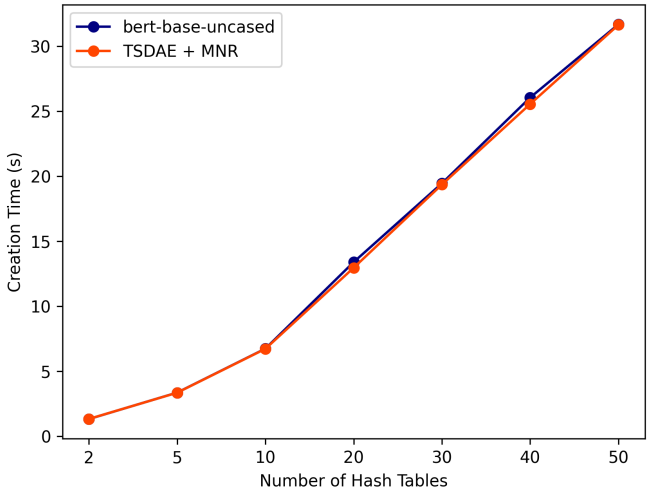
	`bert-base-uncased`		**TSDAE + MNR**	
	API	**Natural Language**	**API**	**Natural Language**
Query ID	**First rank HitRate**	**First rank HitRate**	**First rank HitRate**	**First rank HitRate**
Query 1	2	4	1	1
Query 2	2	0	1	3
Query 3	0	16	1	1
Query 4	0	0	1	12
Query 5	19	20	2	3
Query 6	1	0	2	1
Query 7	14	0	1	0
Query 8	15	0	1	1
Query 9	1	0	2	1
Query 10	0	0	0	5
Query 11	4	0	1	2
Query 12	1	2	1	4
Query 13	0	14	1	1
Query 14	9	0	1	1
Query 15	1	0	1	1
Query 16	0	0	1	1
Query 17	9	10	1	1
Query 18	0	3	1	1
Query 19	12	0	1	1
Query 20	0	0	4	2
MRR	0.29	0.07	0.84	0.71

Tabella 3.2: Risultati della misura di MRR ottenuti per i modelli BERT.

3.3 Average Execution Time

Come mostrato nelle figure 3.1 e 3.2, i tempi di creazione e di raccomandazione per entrambi i modelli sono valutati in base all'Average Execution Time e sono variati in funzione del numero di tabelle hash utilizzate, da un minimo di 2 ad un massimo di 50. Il numero di tabelle hash determina quante volte l'algoritmo di hashing viene applicato sia alla query che ai campioni. Tipicamente, un aumento nel numero di tabelle hash implica una probabilità più alta di mappare un numero maggiore di campioni nello stesso bucket della query. Questo permette di migliorare l'accuratezza della ricerca, consentendo di trovare un numero maggiore di campioni simili. Tuttavia, un aumento delle tabelle hash comporta una maggiore complessità computazionale.

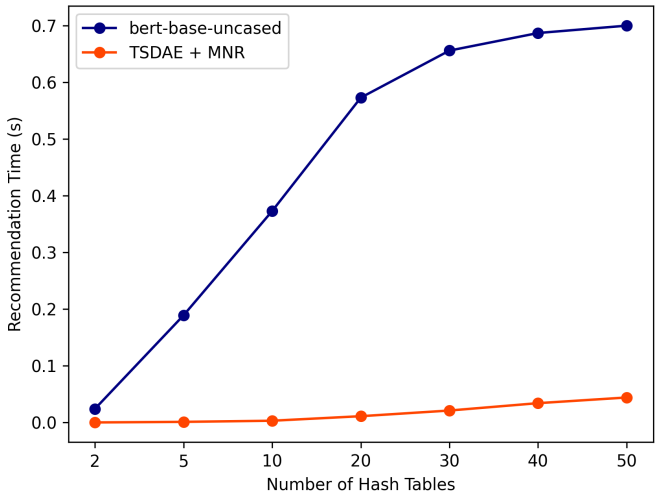
La figura 3.1 illustra i tempi di creazione delle tabelle hash per entrambi i modelli in esame. Dall'analisi dei dati emerge che l'intervallo di valori di questa metrica varia tra 1.325 e 31.681 per il modello `bert-base-uncased`, mentre è compreso tra 1.333 e 31.64 per il modello sottoposto a fine-tuning. Pertanto, si osserva che le prestazioni nel processo di creazione delle tabelle hash per l'algoritmo Query-Aware LSH sono simili per entrambi i modelli, come ci si potrebbe aspettare, e mostrano un andamento lineare con l'aumento del numero di tabelle hash.



	2	5	10	20	30	40	50
bert-base-uncased	1.325	3.36	6.741	13.396	19.455	26.052	31.681
TSDAE + MNR	1.333	3.352	6.726	12.96	19.357	25.552	31.64

Figura 3.1: Tempo di creazione al variare del numero di tabelle hash.

La figura 3.2 illustra i tempi di raccomandazione per entrambi i modelli presi in considerazione. Dall'analisi dei dati, emerge che l'intervallo di valori di questa metrica varia tra 0.024 e 0.7 per il modello `bert-base-uncased`, mentre è compreso tra 0.0 e 0.44 per il modello sottoposto a fine-tuning. Questi dati indicano chiaramente che il modello fine-tuned comporta tempi di raccomandazione migliori rispetto al modello `bert-base-uncased` al crescere del numero di tabelle hash. Tale miglioramento potrebbe essere attribuito al fatto che il modello sottoposto a fine-tuning si è specializzato nel contesto specifico, consentendo una mappatura più efficace dei dati nelle tabelle hash. Di conseguenza, lo spazio di ricerca si riduce notevolmente rispetto al caso del modello base `bert-base-uncased`, portando a tempi di raccomandazione più efficienti.
In particolare, va notato che l'approccio Query-Aware LSH non richiede di allocare tutti i dati all'interno dei bucket. Piuttosto, si focalizza esclusivamente sui campioni il cui valore hash si avvicina a quello della query. Inoltre, l'algoritmo riduce lo spazio di ricerca filtrando i campioni il cui vettore hash è vicino a quello della query durante la fase iniziale del processo di ricerca. Ciò si traduce in una maggiore capacità dell'algoritmo nel riconoscere in modo efficiente i campioni rilevanti.



	2	5	10	20	30	40	50
bert-base-uncased	0.024	0.189	0.373	0.573	0.656	0.687	0.7
TSDAE + MNR	0.0	0.001	0.003	0.011	0.021	0.034	0.044

Figura 3.2: Tempo di raccomandazione al variare del numero di tabelle hash.

3.4 Relevance

La tabella 3.3 riporta i punteggi di Relevance ottenuti dai due modelli esaminati, considerando le prime 20 raccomandazioni restituite ed effettuando query in formato API ed in linguaggio naturale.

	`bert-base-uncased`		**TSDAE + MNR**	
	API	**Natural Language**	**API**	**Natural Language**
Query ID	**Score of Relevance**	**Score of Relevance**	**Score of Relevance**	**Score of Relevance**
Query 1	2	3	2	3
Query 2	3	0	2	3
Query 3	0	2	4	4
Query 4	0	0	4	2
Query 5	1	4	4	3
Query 6	2	0	2	3
Query 7	2	0	4	0
Query 8	4	0	4	4
Query 9	4	0	3	4
Query 10	0	0	0	2
Query 11	4	0	4	3
Query 12	1	4	4	2
Query 13	0	3	2	4
Query 14	3	0	3	3
Query 15	4	0	4	4
Query 16	0	0	3	4
Query 17	4	2	4	4
Query 18	0	2	4	3
Query 19	2	0	2	3
Query 20	0	0	2	2
Average Relevance	1.8	1.0	3.1	3

Tabella 3.3: Risultati della misura di Relevance ottenuti per i modelli BERT.

Gli esempi di codice con un punteggio di rilevanza pari a 3 o 4 sono generalmente considerati utili dagli sviluppatori. In questo contesto, è evidente che il modello sottoposto a fine-tuning presenta un numero significativo di risultati con valori di Relevance pari a 3 o 4, suggerendo che questo algoritmo offre raccomandazioni più pertinenti alle richieste effettuate. Nello specifico, il valore medio della Relevance per il modello fine-tuned è rispettivamente $(3.1, 3)$ per le query in formato API e per le query in linguaggio naturale. Pertanto, tale modello mostra una buona capacità di fornire risposte contestualmente adeguate alle richieste presentate. D'altra parte, le prestazioni del modello `bert-base-uncased` mostrano risultati inferiori, con valori di Relevance medio pari a $(1.8, 1.0)$ rispettivamente per le query API e per le query in linguaggio naturale. Questi risultati indicano che il modello base ha difficoltà a produrre raccomandazioni con un elevato grado di rilevanza, mostrando una limitazione nella comprensione delle richieste fornite dagli sviluppatori, specialmente quando si tratta di query più complesse in linguaggio naturale.

Capitolo 4

Conclusioni

Sulla base dei nostri risultati, emerge che l'algoritmo Query-Aware LSH applicato sul modello sottposto a fine-tuning rappresenta un approccio estremamente efficace per affrontare il problema della raccomandazione di codici di esempio rilevanti in risposta ad una query. Inoltre, sembra che l'utilizzo di query in formato API e in linguaggio naturale non influisca significativamente sui risultati, sottolineando così l'efficacia del processo di fine-tuning precedentemente illustrato. Tuttavia, l'utilizzo di query in formato API mostra una leggera superiorità nel fornire raccomandazioni utili tra le prime posizioni.
Per replicare lo studio effettuato sono stati resi disponibili tutti i dettagli necessari su GitHub.

Bibliografia

[1] Sajjad Rahmani, AmirHossein Naghshzan, and Latifa Guerrouj. Improving code example recommendations on informal documentation using bert and query-aware lsh: A comparative study, 2023.

[2] Stack overflow. Accessed: July 2023.

[3] Database schema documentation for the public data dump and sede. Accessed: July 2023.

[4] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[5] Kexin Wang, Nils Reimers, and Iryna Gurevych. Tsdae: Using transformer-based sequential denoising auto-encoderfor unsupervised sentence embedding learning. *arXiv preprint arXiv:2104.06979*, 4 2021.

[6] Aman Chadha and Vinija Jain. Loss functions. *Distilled AI*, 2020. `https://vinija.ai`.