UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea Magistrale in Informatica

Andrea Bellomo 1000055721

Pre-elaborazione del Codice Java: Oscuramento e Valutazione con un Modello MLM.

Relazione Progetto

Professore : E. Tramontana

Tutor : dott. Midolo

Anno Accademico 2023 - 2024

Indice

Capitolo 1

Introduzione

Nel contesto dell'elaborazione del linguaggio naturale, è possibile esplorare nuove possibilità nella creazione di sistemi intelligenti. I Masked Language Models (MLM) sono una tecnologia chiave in questo ambito, utili a risolvere problemi complicati legati alla comprensione e al trattamento del linguaggio naturale. L'elaborato sviluppato si concentra su come applicare questa tecnologia, realizzando due micro-servizi collegati tra loro per sfruttare al massimo le capacità dei MLM.

L' obiettivo principale è quello di progettare e implementare due servizi autonomi, in grado di interagire in maniera asincrona. Questi servizi saranno dedicati alla manipolazione e all'analisi di codice sorgente Java, utilizzando Masked Language Models per la predizione di porzioni oscurate di codice.

Il primo servizio sarà responsabile di ricevere in input un codice Java e suddividerlo in singoli statement. Successivamente, mediante l'applicazione di una maschera, una porzione specifica di ciascun statement sarà oscurata. Questo processo di oscuramento mira a simulare scenari reali in cui una parte del codice potrebbe essere mancante o non disponibile.

Il secondo servizio riceverà gli statement mascherati dal primo servizio e li sottoporrà a un Modello di MLM, al fine di valutarne l'accuratezza nel predire correttamente il valore mancante. L'implementazione di questa comunicazione asincrona sarà gestita attraverso una coda, dove i dati ricevuti dal primo servizio verranno depositati in attesa di elaborazione.

Capitolo 2

Tecnologie

Ai fini dello sviluppo dell'elaborato, sono stati utilizzati diversi linguaggi di programmazione e tecnologie che saranno, di seguito, descritti nel dettaglio.

2.1 Docker

Docker [1] è una piattaforma open-source progettata per semplificare la creazione, la distribuzione e l'esecuzione di applicazioni in ambienti isolati chiamati container. I container consentono di confezionare un'applicazione e tutte le sue dipendenze in un unico pacchetto, garantendo consistenza e portabilità tra diversi ambienti di sviluppo, test e produzione. Per ciascun microservizio, è stata creata un'immagine Docker che include tutte le dipendenze necessarie, il runtime e le librerie indispensabili per garantire un corretto funzionamento. Questi microservizi comunicano tra loro utilizzando un container RabbitMQ, il quale offre un efficiente sistema di gestione per la coda dei messaggi asincroni. Questa configurazione assicura una comunicazione fluida e affidabile tra i vari microservizi, facilitata dall'uso di RabbitMQ come intermediario per la trasmissione dei messaggi asincroni.

2.2 RabbitMQ

RabbitMQ [2] è un middleware orientato ai messaggi open-source, un broker di messaggistica basato sul protocollo AMQP (Advanced Message Queuing Protocol). Fornisce un efficace meccanismo per la gestione e la distribuzione di messaggi tra applicazioni o microservizi, garantendo una comunicazione asincrona affidabile e scalabile.

Nello specifico, RabbitMQ agisce come intermediario tra il primo microservizio, sviluppato in Java e responsabile della fornitura del codice masche-

rato, e il secondo microservizio, sviluppato in Python e incaricato di applicare il modello di linguaggio mascherato (MLM).

2.3 Masked Language Models (MLM)

I modelli di linguaggio mascherati (MLM) vengono utilizzati nelle attività di elaborazione del linguaggio naturale (NLP) per addestrare modelli linguistici. In questo approccio, alcune parole e token specifici in un input vengono casualmente mascherati o nascosti, e il modello viene quindi addestrato a prevedere questi elementi mascherati utilizzando il contesto fornito dalle parole circostanti.

Il masked language modeling è un tipo di apprendimento auto-supervisionato in cui il modello impara a produrre testo senza etichette o annotazioni esplicite, ottenendo la supervisione direttamente dal testo in ingresso. Grazie a questa caratteristica, il masked language modeling può essere utilizzato per svolgere varie attività di NLP, come la classificazione del testo, la risposta a domande e la generazione di testo.

2.3.1 CodeBERT

CodeBERT [3] è un modello linguistico pre-addestrato per linguaggi di programmazione (Python, Java, JavaScript, PHP, Ruby, Go), sviluppato da Microsoft Research. È progettato per comprendere sia il linguaggio naturale che il codice di programmazione, rendendolo utile per compiti come la sintesi del codice, la traduzione del codice e il completamento del codice. CodeBERT si basa sull'architettura BERT, popolare per i compiti di elaborazione del linguaggio naturale. Attraverso il preaddestramento del modello su un ampio corpus di dati contenenti sia codice che linguaggio naturale, CodeBERT è in grado di apprendere i modelli sintattici e semantici dei linguaggi di programmazione, diventando uno strumento efficace per vari compiti legati al codice.

Esempi di applicazione di CodeBERT:

- **Sintesi del codice**: generare riassunti leggibili dall'uomo di frammenti di codice per facilitare la comprensione e la documentazione del codice.
- **Traduzione del codice**: convertire il codice da un linguaggio di programmazione a un altro, ad esempio tradurre il codice Python in Java.

- **Completamento del codice**: suggerire frammenti di codice per completare il codice scritto parzialmente, migliorando la produttività dello sviluppatore e riducendo la probabilità di errori.

Capitolo 3

Struttura del Sistema

Il sistema proposto si compone da due microservizi che comunicano tra loro tramite un servizio di broker di messaggi RabbitMQ. Di seguito saranno descritte tutte le componenti che compongono il sistema, il quale è stato opportunamente dockerizzato.

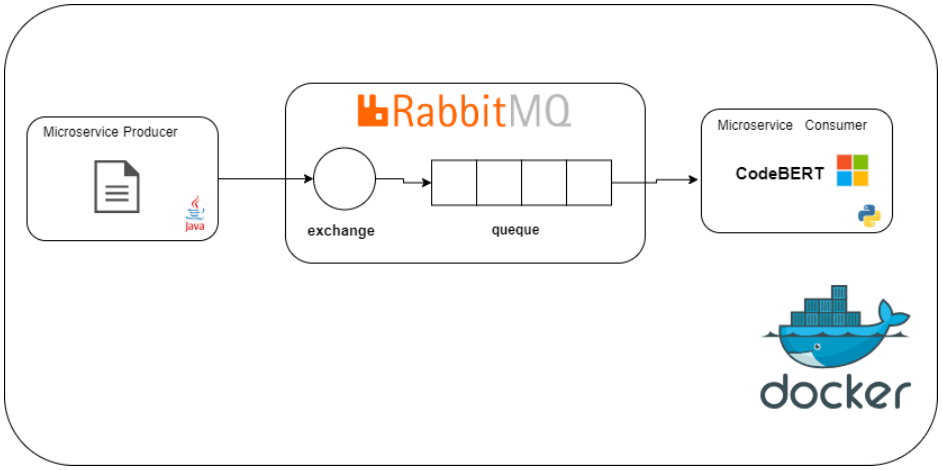

Figura 3.1: Producer

3.1 Java Parser App (Producer)

L'applicazione è composta da tre classi principali: `FileParser`, `RabbitConnect`, e `App`. Queste classi collaborano per leggere un file contenente codice sorgente Java, analizzarlo sintatticamente, applicare una maschera agli statement e inviare i risultati elaborati a una coda RabbitMQ. In seguito verrà descritto il ruolo e la funzionalità di ciascuna classe nel contesto dell'applicazione.

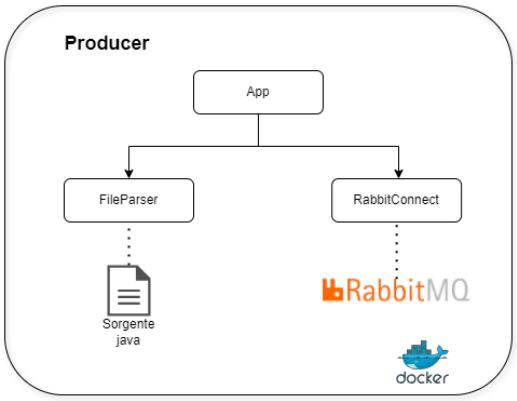

Figura 3.2: Producer

Classe `FileParser`

La classe `FileParser` è dedicata all'analisi sintattica di un file sorgente Java. Implementa un metodo statico `parseFile`, il quale riceve come parametro il percorso del file sorgente. Questo metodo utilizza la libreria JavaParser per convertire il contenuto del file, restituendo una lista di statement presenti nel codice sorgente.

Classe `RabbitConnect`

La classe `RabbitConnect` gestisce la connessione a RabbitMQ. Fornisce metodi statici per la creazione di una connessione, di un canale e per la dichiarazione di una coda specifica. La classe è progettata per semplificare l'interazione con il sistema di messaggistica RabbitMQ, offrendo un'implementazione chiara e riutilizzabile.

Classe `App`

La classe `App` rappresenta l'entry point dell'applicazione. Nel suo metodo `main`, gestisce l intero flusso di esecuzione dell'applicazione. Include la connessione a RabbitMQ, il parsing del file sorgente Java tramite la classe `FileParser`, l'applicazione di una maschera agli statement, e l'invio dei messaggi elaborati alla coda RabbitMQ.

Le tre classi lavorano per fornire un'applicazione completa che legge, analizza e modifica il codice sorgente Java, inviando i risultati elaborati a una coda RabbitMQ. La struttura modulare delle classi favorisce la manutenibilità e la comprensibilità del codice, consentendo una gestione chiara delle respon-

sabilità di ciascuna componente. Questa architettura agevola l'espansione e l'aggiunta di nuove funzionalità all'applicazione in futuro.

3.2 Consumer

Il microservizio che funge da ricevitore è progettato per connettersi a un sistema di messaggistica RabbitMQ. Esso riceve messaggi da una coda specifica, analizzando il contenuto di tali messaggi mediante un modello di linguaggio pre-addestrato CodeBERT,per poi visualizzare i risultati ottenuti. Il microservizio è suddiviso in tre file principali: app.py, consumer.py, e process.py.

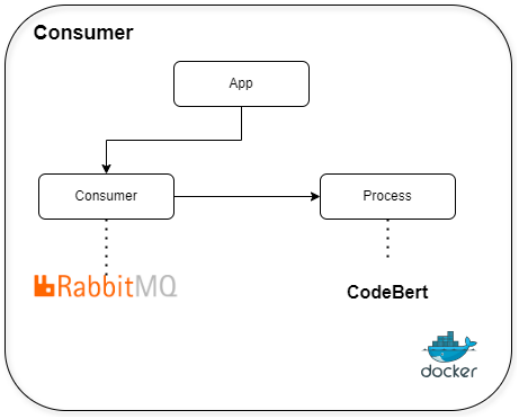

Figura 3.3: Consumer

File `consumer.py`

Il file `consumer.py` contiene la definizione della classe `Consumer`. Questa classe gestisce la connessione a RabbitMQ, il download di un modello di linguaggio e di un tokenizer, nonché l'avvio del consumatore per ricevere i messaggi dalla coda 'my-queue'. Quando un messaggio viene ricevuto, la classe chiama il metodo `callback` per decodificarlo e avviare la classificazione utilizzando la classe `classifier` del modulo `process`.

File `process.py`

Il file `process.py` contiene la definizione della classe `classifier`, che utilizza la libreria Transformers di Hugging Face. La classe `classifier`

carica un modello pre-addestrato di tipo Roberta per il completamento di maschere e un tokenizzatore associato. Il metodo `start_classifier` prende un messaggio in input, predice una maschera mancante e stampa i risultati ottenuti.

Comportamento Generale

1. **Inizializzazione e Connessione:** Il microservizio inizia con l'esecuzione di `app.py`. Questo file crea un'istanza della classe `Consumer` e avvia la connessione a RabbitMQ.

2. **Consumo di Messaggi:** Il consumatore (`consumer.py`) è responsabile di ascoltare la coda 'my-queue' su RabbitMQ e di gestire i messaggi ricevuti chiamando il metodo `callback`.

3. **Elaborazione del Messaggio:** Il metodo `callback` decodifica il messaggio, avvia la classificazione utilizzando la classe `classifier` del modulo `process.py`, e quindi conferma l'avvenuta elaborazione del messaggio a RabbitMQ.

4. **Classificazione del Messaggio:** La classe `classifier` utilizza un modello pre-addestrato per predire una maschera mancante all'interno del messaggio e stampa i risultati ottenuti.

La suddivisione in file separati favorisce la modularità e la manutenibilità del codice.

Capitolo 4

Configurazione del Sistema

Questa sezione fornisce una panoramica completa della configurazione del sistema, che comprende l'orchestrazione dei microservizi mediante Docker Compose e l'interazione tra di essi.

4.1 DockerFile Consumer

Il Dockerfile utilizza un'immagine ufficiale di Python versione 3.9-slim come base. La directory di lavoro (WORKDIR) viene impostata su /app, dove vengono copiati tutti i file necessari per l'esecuzione del microservizio. Successivamente, vengono installate le dipendenze specificate nel file requirements.txt del microservizio Consumer. Infine, l'istruzione CMD specifica il comando predefinito che viene eseguito quando il container è avviato, eseguendo il file app.py del microservizio Consumer.

4.2 DockerFile Producer

Il Dockerfile utilizza un'immagine base con il supporto per Java, specificamente l'immagine maven:3.8.4-jdk-11-slim. La directory di lavoro (WORKDIR) viene impostata su /producer. Successivamente, il file pom.xml necessario per la compilazione del progetto viene copiato nella directory di lavoro, seguito dalla copia dei file restanti della cartella producer. Il progetto viene quindi compilato utilizzando Maven e, infine, l'istruzione CMD definisce l'entry point dell'applicazione, eseguendo il file JAR generato.

4.3 Docker Compose

Analisi del Docker Compose

Il Docker Compose fornito definisce la configurazione per l'orchestrazione dei servizi relativi al sistema. Di seguito è riportata una relazione dettagliata sulla struttura e la funzionalità di questo file.

1. **Versione Docker Compose:** Il file utilizza la versione '3' del formato Docker Compose.
2. **Servizio RabbitMQ (rabbitmq-container):**
   - **Immagine:** L'immagine utilizzata è "rabbitmq:3-management" che include il supporto alla gestione tramite interfaccia web.
   - **Porte:** Le porte 5672 e 15672 del container sono mappate rispettivamente alle porte 5672 e 15672 dell'host.
   - **Networks:** Il servizio è associato alla rete denominata 'mynet'.
   - **Healthcheck:** Viene eseguito un health check per verificare lo stato di RabbitMQ tramite il comando "rabbitmqctl status".
   - **Command:** Viene specificato il comando "rabbitmq-server".
3. **Servizio Java App (java-app-container):**
   - **Build:** Il servizio viene costruito utilizzando il Dockerfile situato nella directory `./producer`.
   - **Networks:** Il servizio è associato alla stessa rete 'mynet' di RabbitMQ.
   - **Depends On:** Il servizio dipende da 'rabbitmq-container', aspettando che risulti sano prima di essere avviato.
4. **Servizio Receiver (receiver-container):**
   - **Build:** Il servizio viene costruito utilizzando il Dockerfile situato nella directory `./consumer`.
   - **Networks:** Il servizio è associato alla stessa rete 'mynet'.
   - **Depends On:** Il servizio dipende da 'rabbitmq-container', aspettando che risulti sano prima di essere avviato.
5. **Networks (mynet):**
   - **Driver:** La rete 'mynet' è una rete di tipo bridge.

Capitolo 5

Risultati

Al fine di testare l'intero sistema è possibile utilizzare qualsiasi codice sorgente Java. Al fine di valutare le prestazioni del sistema sono stati generati diversi esempi di codice tramite ChatGPT.

Il codice sorgente da analizzare è stato inserito nel file sorgente.txt e successivamente analizzato.

Di seguito verranno descritti alcuni statement con lo scopo di comprendere il comportamento del modello su token corretti o errati. Successivamente sarà presentata una tabella riassuntiva con la percentuale media di accuratezza.

5.1 Demo

Nella seguente sezione sono mostrati diversi esempi di predizione. Dove X indica lo statement con il token mascherato e Y indica il token che è stato mascherato e che deve essere predetto. Il token predetto indica il token che è stato predetto da CodeBERT.

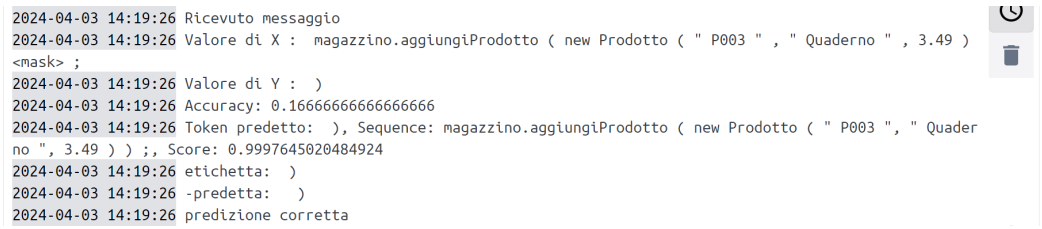

Figura 5.1: esempio predizione corretta 1

```
2024-04-03 15:38:15 Valore di X :  this.value = <mask> ;
2024-04-03 15:38:15 Valore di Y :  value
2024-04-03 15:38:15 Accuracy: 0.5547445255474452
2024-04-03 15:38:15 Token predetto:  value, Sequence: this.value = value ;, Score: 0.8285128474235535
2024-04-03 15:38:15 etichetta:  value
2024-04-03 15:38:15 -predetta:   value
2024-04-03 15:38:15 predizione corretta
```

Figura 5.2: esempio predizione corretta 2

```
2024-04-03 15:37:44 Ricevuto messaggio
2024-04-03 15:37:44 Valore di X :  <mask> ;
2024-04-03 15:37:44 Valore di Y :  break
2024-04-03 15:37:44 Accuracy: 0.5257731958762887
2024-04-03 15:37:45 Token predetto:  ), Sequence:  ) ;, Score: 0.26295310258865356
2024-04-03 15:37:45 etichetta:  break
2024-04-03 15:37:45 -predetta:   )
```

Figura 5.3: esempio predizione errata 1

```
2024-04-03 15:37:33 Ricevuto messaggio
2024-04-03 15:37:33 Valore di X :  <mask> score ;
2024-04-03 15:37:33 Valore di Y :  return
2024-04-03 15:37:33 Accuracy: 0.5542168674698795
2024-04-03 15:37:34 Token predetto:  =, Sequence:  = score ;, Score: 0.5774964690208435
2024-04-03 15:37:34 etichetta:  return
2024-04-03 15:37:34 -predetta:   =
```

Figura 5.4: esempio predizione errata 2

Il modello è stato testato su diversi codici generati con diversi livelli di complessità.

Codice	**Accuracy (%)**
generated code 1	54,83%
generated code 2	50%
generated code 3	56,11%

Tabella 5.1: tabella riassuntiva

Possiamo notare come il modello CodeBERT usato per fare inferenza riesce a predire in media il **53%** dei token mascherati.

5.2 Sviluppi futuri

Di seguito sono elencati alcuni possibili sviluppi futuri:

- Effettuare il Fine Tuning di CodeBERT considerarado l'integrazione di altre fonti di dati, per migliorare la diversità e la ricchezza dei dati di addestramento. Questo potrebbe includere l'uso di dati aggiuntivi da repository di codice open source.
- Valutare le Prestazioni su Dati Reali al fine di fornire una comprensione più accurata dell'efficacia del modello in scenari pratici e consentire eventuali ottimizzazioni aggiuntive.
- Esplorare la possibilità di implementare funzionalità aggiuntive nel sistema, come la correzione automatica dei bug.

Bibliografia

[1] Inc. Docker. Docker: Platform for building, shipping, and running applications. https://www.docker.com/, 2013.

[2] Inc. Pivotal Software. Rabbitmq: Messaging that just works. https://www.rabbitmq.com/, 2007.

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.