PC-2023/24 Bigram/Trigram Calculator

Alessandro Filino
E-mail address
alessandro.filino@edu.unifi.it

Abstract

Questo progetto si focalizza sull'implementazione di un sistema in grado di identificare e numerare bigrammi e trigrammi presenti all'interno di un dataset di libri. Conclusa l'analisi di ogni libro viene quindi creata una statistica generale.

1. Introduzione

L'analisi del linguaggio naturale si avvale di tecniche come i bigram e i trigram per comprendere le relazioni tra le parole in un testo. Questi approcci sono utilizzati in diversi ambiti e per diversi scopi come la correzione automatica degli errori in un testo.

1.1. Framework e Linguaggi di Programmazione

Lo sviluppo si è inizialmente concentrato in una implementazione sequenziale attraverso il linguaggio Python [Sez. 2].
Successivamente ci siamo concentrati su una implementazione parallela seguendo il modulo multiprocessing fornito da Python stessa.
Multiprocessing permette infatti di eseguire sequenze di codice in parallelo anche in Python aggirando il problema del GIL (ogni processo lanciato da multiprocessing ha un proprio spazio di memoria separato e quindi non condivide lo stesso GIL con gli altri processi).
Infine, per come è stato posto il problema, si è cercato una soluzione ancora più performante che sfrutti sia multiprocessing che asyncio.

1.2. Strumenti di sviluppo

Per la realizzazione del progetto sono stati utilizzati i seguenti tool in fase di sviluppo:

- **Visual Studio Code**: IDE sviluppato da Microsoft [3], ampiamente utilizzato per lo sviluppo di una vasta gamma di linguaggi di programmazione, incluso Python. VS Code offre un'ampia gamma di estensioni e tool integrati che semplificano lo sviluppo e migliorano la produttività generale. L'utilizzo di visual studio con Python offre alcune features com l'autocompletamento, la formattazione del codice, il debugging, la gestione dei virtual environment e la gestione di più progetti contemporaneamente
- **Project Gutenberg**: Project Gutenberg [2] è una libreria digitale che offre accesso gratuito a migliaia di opere letterarie di pubblico dominio.
  Viene fondata nel 1971 con l'obiettivo di digitalizzare, archiviare e rendere disponibili libri in varie lingue e generi.
  Inoltre offre la possibilità di selezionare il formato con cui scaricare il documento tra cui: ePub, TXT, PDF e HTML

2. Implementazione

2.1. Downloader

Per trovare un ampio dataset di libri analizzabili, è stata sfruttata la libreria pubblica di Gutenberg [2]. In particolare, specificato il numero totale di libri da analizzare, la funzione **download_gutenberg_book** compone URI in questo

formato:

base_url / book_id . book_format

dove:

- base_url: Identifica l'url standard per accedere alla sezione di ebook di Gutenberg
- book_id: Identifica l'i-esimo libro da scaricare
- book_format: Identifica il formato da utilizzare. Nel nostro caso useremo `.txt.utf-8` per ottenere file in formato `.txt`

Quindi se volessimo scaricare il libro che all'interno del sito è identificato con l'indice 125, il link si comporrà in questo modo: *https://www.gutenberg.org/ebooks/125.txt.utf-8*

La funziona si occuperà inoltre dell' organizzazione dei vari download all' interno della directory *resources* rinominando il file scaricato di ogni libro nel formato `book_n` (dove **n** indica l'indice del libro) e la gestione di eventuali errori di download per evitare l'interruzione dell'intero codice.

2.2. NTLK - Natural Language Toolkit

Per trovare all'interno di un testo la sequenza di bigrammi e trigrammi, l'implementazione si basa sull'utilizzo della libreria **NTLK** [4].
NLTK è una delle librerie più popolari in Python per il processing del linguaggio naturale (*Natural Language Toolkit*) e offre la possibilità di eseguire analisi di n-gram di parole.
Il testo prima di essere classificato da NLTK segue una fase di preprocessing (**preprocess_text(***text***)**) in cui vengono rimossi i caratteri di tipo non alfanumerico ed eventuali spazi tra parole attraverso l'uso di una espressione regolare: `r'[^\w\s]'`.
In questo modo evitiamo di analizzare stringhe di caratteri comprensibili solo dallo standard ASCII (e quindi inutili nel linguaggio naturale) o coppie formate da uno spazio e una parola (*es.* `' mamma'`).
Dopo aver ottenuto una lista di bigrammi e trigrammi, questa viene passata alla funzione **FreqDist()** messa a disposizione da NTLK. Questa funzione crea una mappa che ha per chiave il bigramma/trigramma in analisi e per valore il numero di occorrenze trovate all'interno della lista ottenuta in precedenza.
Il risultato dell'analisi di ogni libro costituirà un risultato parziale per l'analisi complessiva. Questo viene quindi aggiunto alla rispettiva coda condivisa: *bigram_shared_queue* e *trigrams_shared_queue*.

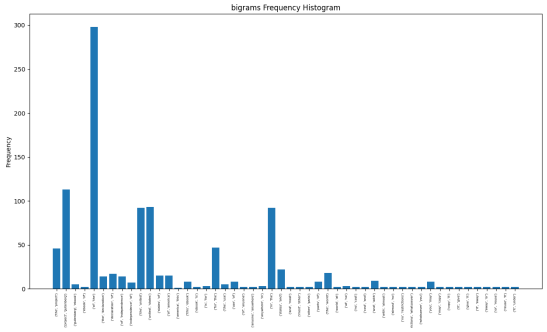

Figure 1. Analysis from Book 1: Bigram Example (limit at 50)

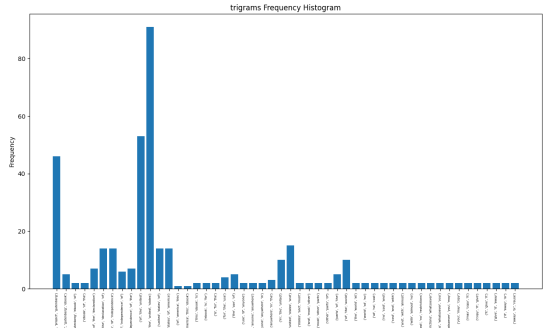

Figure 2. Analysis from Book 1: Trigram Example (limit at 50)

Concluse tutte le analisi parziali, dovremmo quindi produrre un risultato generale dei dati raccolti. Per fare questo, attraverso la funzione **combine_result** ordineremo i risultati parziali trovati producendo una mappa che ha per chiavi le coppie/triple di parole trovate senza ripetizioni e per valori la somma delle occorrenze trovate per ogni libro.
Viene infine prodotto un file in formato Json con i risultati trovati e un istogramma generale rappre-

sentativo.

Listing 1. combine_and_log_results

```
result_dict = {}
while not shared_queue.empty():
    item = shared_queue.get()
    for key, value in item.items():
        result_dict[key] =
            result_dict.get(key, 0) +
            value
log_to_file(working_directory,
    result_dict, file_name)
```

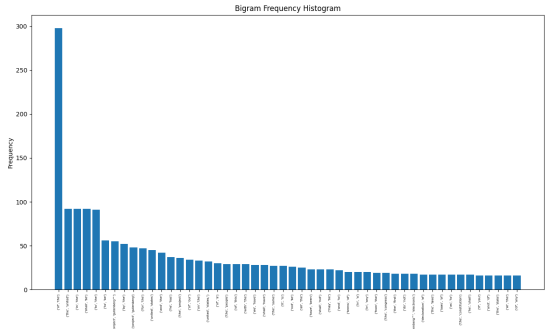

Figure 3. Analysis of 100 books: Bigram Example (limit at 50)

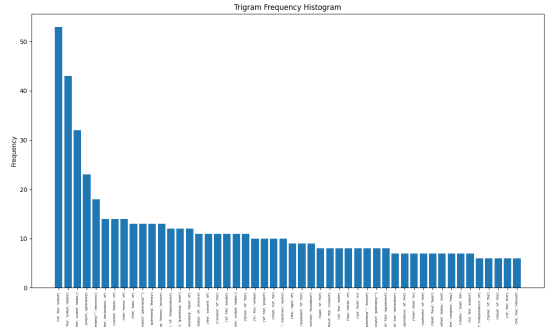

Figure 4. Analysis of 100 books: Trigram Example (limit at 50)

3. Parallelization

3.1. Multiprocessing

Il primo approccio utilizzato per parallelizzare il codice è stato quello di sfruttare il modulo Multiprocessing offerto da Python [5].
L'obiettivo è quello di distribuire il numero di libri da elaborare ad un determinato numero di processi. Ogni processo è quindi responsabile di scaricare un libro ed eseguirne l'elaborazione parziale dei bigrammi e trigrammi:

```
with Pool(processes=processes_number) as
    process_pool:
    process_pool.map(process_book, args)
```

Conclusa questa prima parte, i risultati ottenuti saranno inseriti in una coda condivisa tra tutti i processi. Nello specifico questo è stato fatto sfruttando il meccanismo di coda già presente nel modulo Multiprocessing che si occuperà dell' accesso e della sincronizzazione: `Manager().Queue()`.

Concluse tutte le elaborazioni parziali dei vari libri, vengono lanciati due processi distinti che in parallelo:

- Uno si occuperà di produrre il dizionario complessivo per i bigrammi attraverso l'analisi dei dati raccolti nella coda condivisa. Dopodiché produrrà il file Json contenente i risultati
- Uno identico al precedente ma per la gestione dei trigrammi

```
bigrams_process =
        Process(target=combine_and_log_results,
                args=(bigrams_shared_queue,
                    "bigrams"))
trigrams_process =
        Process(target=combine_and_log_results,
                args=(trigrams_shared_queue,
                    "trigrams"))
```

3.2. Multiprocessing e Asyncio

Per ottimizzare ulteriormente le performance del nostro codice, essendo la parte iniziale principalmente di tipo I/O bound, era lecito aspettarsi l'utilizzo di Asyncio.
In questo modo l'esecuzione prevede:

- Una prima parte in cui vengono scaricati tutti i libri richiesti
- Successivamente, i libri vengono elaborati in maniera parallela attraverso Multiprocessing come spiegato nella Sez. 3.1

Per la prima parte è stato utilizzato il modulo *aiohttp* [1] ridefinendo la funzione **download_gutenberg_book** e richiamandola in questo modo:

```
async with aiohttp.ClientSession() as
    session:
```

```
download_and_process_tasks = [
download_gutenberg_book(working_director
    session, book)
        for book in range(books)
]
await
 asyncio.gather(*download_and_process
                _tasks)
```

La funzione **download_gutenberg_book** si occuperà quindi di creare sessioni asincrone http, eseguire le richieste GET di download e verificarne lo stato di risposta da parte del server web.

4. Test and Results

Andiamo adesso ad analizzare i risultati ottenuti dalle varie esecuzioni.
Per prima cosa abbiamo valutato in generale l'andamento dei tre approcci all'aumentare del numero di libri da analizzare.

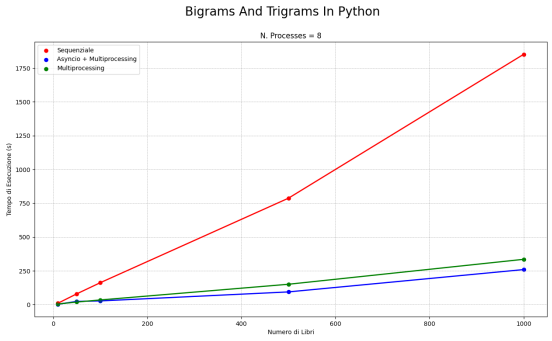

Figure 5. Confronto Generale: Sequenziale, Multiprocessing, Multiprocessing + Asyncio

Come possiamo vedere in Fig. 5, il metodo sequenziale (riportato in rosso) risulta ovviamente essere quello meno performante mentre i due paralleli, multiprocessing (in verde) e multiprocessing + asyncio (in blu), risultano essere molto simili utilizzando in entrambi i casi 8 processi.

Successivamente abbiamo supposto di dover analizzare un dataset composto da 100 libri e abbiamo valutato le esecuzioni variando il numero di *processes* utilizzati nella parte di multiprocessing dei due approcci paralleli.

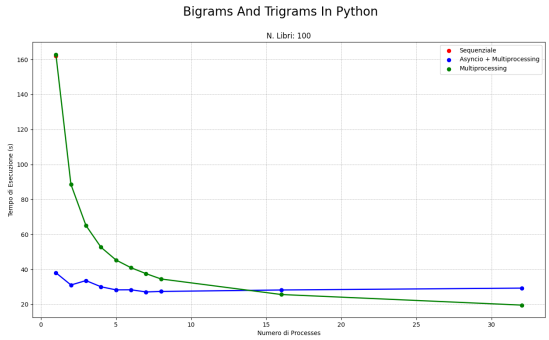

Figure 6. Confronto 100 libri e n_processes: Sequenziale, Multiprocessing, Multiprocessing + Asyncio

Il risultato ottenuto mostra come inizialmente, a parità di *processes*, asyncio fornisca un ottimo boost di prestazioni iniziali anche se paragonato con multiprocessing.
Valutando anche lo **speed-up**, ossia il rapporto tra il tempo di esecuzione del sistema sequenziale e quello parallelo, possiamo notare come in asyncio (Fig. 7) la curva ottenuta sia molto ripida inizialmente per poi attenuarsi con l'aumentare del numero dei processi.

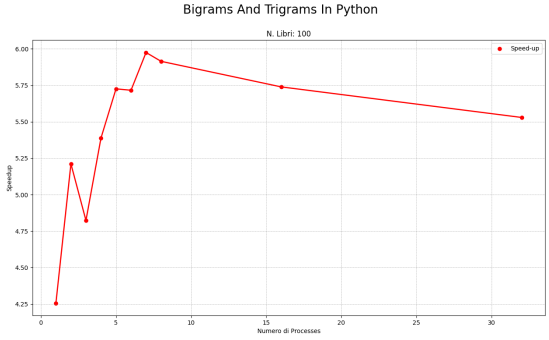

Figure 7. Speed-up 100 libri e n_processes: Sequenziale, Multiprocessing + Asyncio

Valutando invece lo speed-up con multiprocessing (Fig. 8), la curva ottenuta è meno ripida ma più costante aumentando il numero di processi in uso.

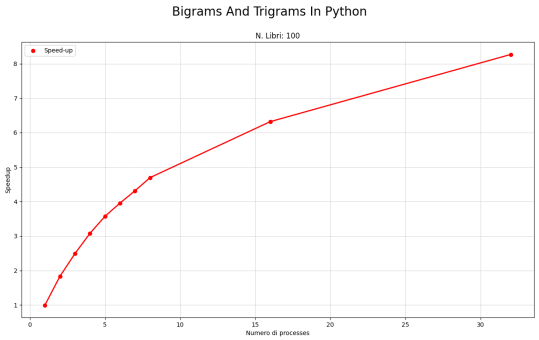

Figure 8. Speed-up 100 libri e n_processes: Sequenziale, Multiprocessing

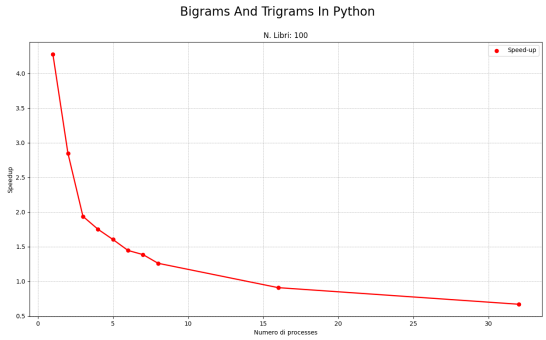

Figure 9. Speed-up 100 libri e n_processes: Multiprocessing / Asyncio

Tornando alla Fig. 6, possiamo notare come aumentando il numero di processi (attorno a 14 processi paralleli) otteniamo che l' approccio con multiprocessing risulta essere più efficiente di utilizzare asyncio + multiprocessing.
Questo ci fa intuire come in questo tipo di problema risulti più vantaggioso aumentare il numero di *processes* (tenendo conto dell'hardware a disposizione) piuttosto che attendere il completamento di un processo asincrono di tipo I/O.
La spiegazione può essere valida facendo alcune considerazioni:

- Il download è stato effettuato utilizzando una connessione FTTC. Durante i test infatti la banda di download veniva saturata completamente. Una maggiore larghezza di banda a disposizione potrebbe rendere asyncio più efficiente poiché si abbasserebbe il rischio di saturazione
- Il download è stato effettuato attraverso i server di Gutenberg. Dovremmo quindi valutare la banda messa a disposizione da essi nonché il numero massimo di sessioni che possiamo aprire

Analizzando infatti lo speed-up tra i due approcci paralleli (Fig. 9), otteniamo la conferma a quanto detto ossia come asyncio fornisca un ottimo boost di prestazioni iniziali per il download dei libri ma come, aumentando il numero processi attivi, sia più efficiente assegnare ad ogni processo il download e l'elaborazione di un libro piuttosto che attendere inizialmente il completamento del processo di download asincrono:

References

[1] aiohttp contributors. Asyncio - AIOHTTP. `https://docs.aiohttp.org`.
[2] Gutenberg. Project Gutenberg. `https://www.gutenberg.org`.
[3] Microsoft. Visual Studio Code. `https://code.visualstudio.com`.
[4] NLTK Project. NLTK. `https://www.nltk.org`.
[5] Python Software Foundation. Multiprocessing. `https://docs.python.org/3/library/multiprocessing.html`.