UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Tesi di Laurea Triennale in Informatica

Utilizzo degli LLM per Generare Hugging Face Model Card da esempi di codice

Relatore
Dr. Di Rocco Juri

Correlatore
Dr. Di Sipio Claudio

Laureando
Stefano Palombo
278930

Anno Accademico 2023-2024

Abstract

Negli ultimi anni, il rapido sviluppo del machine learning ha portato alla creazione di modelli sempre più avanzati e complessi, rendendo essenziale la disponibilità di strumenti che ne facilitino l'accesso e l'integrazione da parte degli sviluppatori.
Tra le piattaforme più influenti in questo ambito, Hugging Face si distingue per la sua vasta collezione di modelli open-source pre-addestrati (Pre-Trained Models, PTM) e per una community attiva nello sviluppo e documentazione dei modelli stessi. Ogni modello pubblicato è corredato da una model card, un documento che ne descrive le caratteristiche tecniche e i possibili casi d'uso, spesso accompagnato da esempi di codice Python per agevolarne l'implementazione.
L'obiettivo di questa tesi è analizzare la corrispondenza tra il codice presente nelle model card ufficiali di Hugging Face e l'effettivo utilizzo dei modelli in progetti open-source su GitHub. Per farlo, è stato adottato un approccio basato su data-mining, analisi del codice e generazione automatica di pattern mediante un LLM (Large Language Model).
Il primo passo è stato la raccolta e l'analisi di 141,749 script Python provenienti da repository pubblici, con lo scopo di estrarre file di codice che mostrassero l'uso pratico di 1024 modelli di Hugging Face. Questi snippet sono stati poi clusterizzati e filtrati per individuare pattern ricorrenti, che successivamente sono stati sintetizzati da un LLM, producendo 875 frammenti di codice rappresentativi. Parallelamente, è stato creato un dataset con gli snippet di codice presenti nelle model card ufficiali di Hugging Face, coprendo 751 modelli. Dopo aver intersecato questi dati con quelli prodotti dall'LLM, le metriche di similarità, comunemente usate in letteratura, sono state calcolate per 625 modelli comuni.
Le analisi hanno evidenziato una similirità parziale pari al circa 40% dimostrando che il codice generato dall'LLM non è una semplice copia del codice ufficiale, ma una sintesi alternativa basata su pattern più specifici, estratti direttamente dagli snippet reali.
Sebbene le metriche automatiche possano non sempre cogliere le somiglianze a livello concettuale, il codice generato dall'LLM può essere comunque utile per gli sviluppatori umani, che, attraverso un'analisi manuale, sono in grado di individuare dettagli tecnici rilevanti. Inoltre, per i modelli che non dispongono di esempi di codice nelle loro model card ufficiali, l'output generato dall'LLM potrebbe essere utilizzato come base per arricchire la documentazione, facilitando l'adozione dei modelli all'interno della community.
L'approccio sviluppato in questa tesi potrebbe, quindi, costituire un primo passo verso un sistema automatico di documentazione, in grado di arricchire le model card esistenti e fornire informazioni più dettagliate e contestualizzate sull'utilizzo reale dei modelli di Hugging Face, migliorando così l'accessibilità e la qualità.

Keywords: PTM, Hugging Face, model card, data-mining, pattern, clustering, LLM, metriche.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Negli ultimi anni, il campo dell'intelligenza artificiale e, in particolare del machine learning, ha subito una crescita esponenziale portando allo sviluppo di modelli sempre più complessi e performanti. Questo progresso ha reso necessario non soltanto il perfezionamento delle architetture dei modelli, ma anche la realizzazione di strumenti che ne facilitano l'accesso e l'implementazione da parte della community degli sviluppatori

Tra le varie piattaforme che rendono questi modelli più accessibili emerge *Hugging Face*[1], che offre agli sviluppatori, tramite librerie open-source, la possibilità di sperimentarli per una vasta gamma di attività che spaziano tra la generazione e classificazione di testo naturale, classificazione di immagini, rilevazione di oggetti e molto altro.

L'utilità di Hugging Face non risiede soltanto nella disponibilità di modelli avanzati, ma anche nella sua community attiva, che contribuisce non solo nello sviluppo, ma anche alla documentazione dei modelli stessi. Ogni modello pubblicato sulla piattaforma è infatti corredato da una *model card*, un documento che ne descrive le caratteristiche principali, le limitazioni e i possibili casi d'uso, mostrando esempi di codice in linguaggio Python per facilitarne l'integrazione in applicazioni reali. L'esperimento di tesi si colloca proprio in questo contesto, concentrandosi sull'analisi dell'effettivo utilizzo dei modelli di Hugging Face nei progetti open-source.

L'obbietivo principale è misurare quanto il codice presente sulle model card di Hugging Face sia effettivamente simile all'utilizzo reale del modello in progetti software, presenti sulla piattaforma Github[2]. Per rispondere a questa domanda è stato adottato un approccio che si basa sull'estrazione automatica di *pattern* ricorrenti di codice Python, ossia sequenze di funzioni e chiamate che mostrano il flusso tipico di un modello: dal caricamento iniziale, alla preparazione dei dati, fino alle fasi di addestramento e inferenza.

Per raggiungere questo scopo è stata inizialmente condotta una ricerca approfondita, tramite *data-mining*, degli script presenti nei repository GitHub che presentassero un riferimento ai modelli di Hugging Face, al fine di costruire un dataset strutturato di file che contenessero effettivamente codice utilizzato dalla community. Successivamente, questi script sono stati sottoposti a un'analisi automatizzata per individuare le porzioni semanticamente rilevanti, utilizzando metodi di parsing del codice e tecniche di *clustering*, per poi servirsi di un LLM (*Llama 3.2*[3]) per generare una versione sintetizzata e rappresentativa delle porzioni di codice individuate. Per quantificare il grado di

[1] `https://huggingface.co/`
[2] https://github.com/
[3] https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct

somiglianza tra il codice delle model card presenti sulla piattaforma e quello effettivamente prodotto dall'approccio, si sono impiegate metriche di similarità del codice, come CodeBLEU[22], Cosine Similarity e METEOR[2].
L'estrazione e l'analisi di queste sequenze di codice forniscono informazioni utili non solo per valutare la coerenza tra documentazione ufficiale e utilizzo pratico, ma anche per comprendere meglio come i modelli vengono adottati nella realtà. L'analisi dei pattern permette, infatti, di identificare best practices, iperparametri specifici che potrebbero non essere immediatamente evidenti nelle model card.
L'approccio sviluppato potrebbe anche essere utilizzato in futuro per arricchire automaticamente le model card che attualmente non contengono esempi di codice, contribuendo a migliorare la qualità e la fruibilità della documentazione dei modelli preaddestrati, rendendo l'ecosistema di Hugging Face ancora più accessibile.
In particolare, il lavoro è articolato in diverse sezioni, ognuna delle quali approfondisce un aspetto specifico dell'approccio adottato per l'analisi dell'utilizzo dei Pre-trained Models (PTM) nei progetti open-source.
Nel Capitolo *Contesto e motivazione*, vengono introdotte le basi teoriche degli strumenti utilizzati nell'esperimento, con una panoramica su concetti fondamentali del funzionamento degli LLM e clustering. Viene inoltre evidenziata la necessità di sviluppare un approccio automatizzato per l'estrazione di pattern di utilizzo dei modelli, con particolare attenzione alla documentazione di Hugging Face che non contiene esempi di codice Python.
Successivamente, il Capitolo *Approccio* descrive il flusso di lavoro adottato per selezionare, analizzare e processare gli snippet di codice. Vengono dettagliate le tecniche per individuare pattern ricorrenti e il ruolo dell'LLM nella sintesi automatica del codice.
Nella sezione *Validazione e Risultati*, vengono presentate le metriche adottate per confrontare il codice generato dall'LLM con gli esempi presenti nelle model card ufficiali di Hugging Face. Si discutono i risultati ottenuti e l'efficacia dell'approccio nel riprodurre gli schemi di utilizzo dei modelli.
A seguire, la sezione *Limitazioni* analizza i principali vincoli incontrati nel processo, tra cui le difficoltà legate alla selezione dei modelli, la qualità dell'LLM impiegato, le restrizioni delle API utilizzate e le criticità nella costruzione del dataset di test.
Infine, nella sezione *Conclusioni e Sviluppi Futuri*, viene proposto un riepilogo dei risultati ottenuti e vengono illustrate possibili evoluzioni dell'approccio, come il miglioramento delle tecniche di estrazione degli snippet e l'estensione della generazione automatica dell'intera model card per colmare le lacune nella documentazione dei modelli di Hugging Face.
Il dataset prodotto nella fase di data-mining, contenente gli scipt Python, è visionabile gratuitamente [8]. Inoltre questi dati sono stati usati per una pubblicazione[5] che è stata inizialmente rigettata, ma su cui si sta attualmente lavorando per migliorarla.
Invece, il codice prodotto è disponibile su richiesta sulla piattaforma Github[4].

[4] `https://github.com/stefanopalombo8/tesi_triennale`

Capitolo 2

Contesto e motivazione

In questa sezione verranno illustrati i principali strumenti impiegati, evidenziando il loro ruolo all'interno del workflow sperimentale e il motivo della loro scelta per ottimizzare l'accuratezza e l'efficienza dell'analisi:

2.1 Hugging Face Model card

Rappresenta la documentazione ufficiale di un Pre-trained Models (PTM) e include informazioni tecniche e pratiche per il suo utilizzo[1]. Ogni model card è associata a un repository specifico su Hugging Face e solitamente è contenuta in un file README.md, accessibile direttamente dalla pagina web del modello (es. in Figura 2.1 corrispondente alla card[2]).
Le informazioni comprese in una model card variano a seconda del creatore del modello, ma generalmente comprendono una descrizione generale, che illustra l'architettura del modello, lo scopo e le principali caratteristiche. Vengono poi specificati i task supportati, come classificazione, generazione di testo, embedding e traduzione. Inoltre, viene indicata la licenza sotto cui il modello è rilasciato, un aspetto fondamentale per comprenderne le condizioni d'uso. Un altro elemento chiave è la descrizione dei dataset di pre-training, che fornisce informazioni sulle fonti dei dati utilizzati per l'addestramento del modello. Se presenti, vengono segnalate eventuali restrizioni sulla lunghezza massima del contesto supportato, comunemente indicate come token limit.
Una parte della model card potrebbe essere dedicata alla modalità di fine-tuning, offrendo suggerimenti su come adattare il modello per compiti specifici. Sono, inoltre, fornite informazioni su inferenza e deployment, con esempi pratici e spesso accompagnati da snippet di codice Python che utilizzano la libreria open-source *transformers* [30], sviluppata direttamente da Hugging Face. Infine, vengono evidenziate le principali limitazioni del modello, incluse le performance su dati specifici e i potenziali problemi legati al suo utilizzo.
Oltre al contenuto testuale, le model card includono una sezione di metadati in formato YAML all'inizio del file README.md. Questi metadati facilitano la ricerca e il filtraggio dei modelli nel Model Hub, consentendo agli utenti di selezionare modelli in base a criteri come licenza, dataset utilizzati e lingue supportate.
Hugging Face fornisce diverse risorse per aiutare gli sviluppatori a compilare correttamente una

[1] `https://huggingface.co/docs/hub/model-cards`
[2] `https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2`

model card tra cui un template con sezioni predefinite e una guida interattiva.
Invece quelle già pubblicate possono essere aggiornate e modificate solo da utenti autorizzati come creatori del modello e gli amministratori di Hugging Face. Tuttavia, la community può proporre può proporre modifiche direttamente al file README.md, sia tramite l'interfaccia web di Hugging Face, sia utilizzando il repository Git associato.
In generale, le model card svolgono un ruolo fondamentale nella condivisione e nell'utilizzo responsabile dei modelli, garantendo trasparenza e accessibilità alle informazioni tecniche a benificio delle persone anche con background diversi dall'IT.
Infine, recenti studi su larga scala hanno analizzato le model card esistenti, evidenziando che sezioni come impatto ambientale, limitazioni e valutazioni sono spesso poco documentate, mentre le informazioni sull'addestramento tendono a essere più complete [13]. Nella descrizione del lavoro suc-

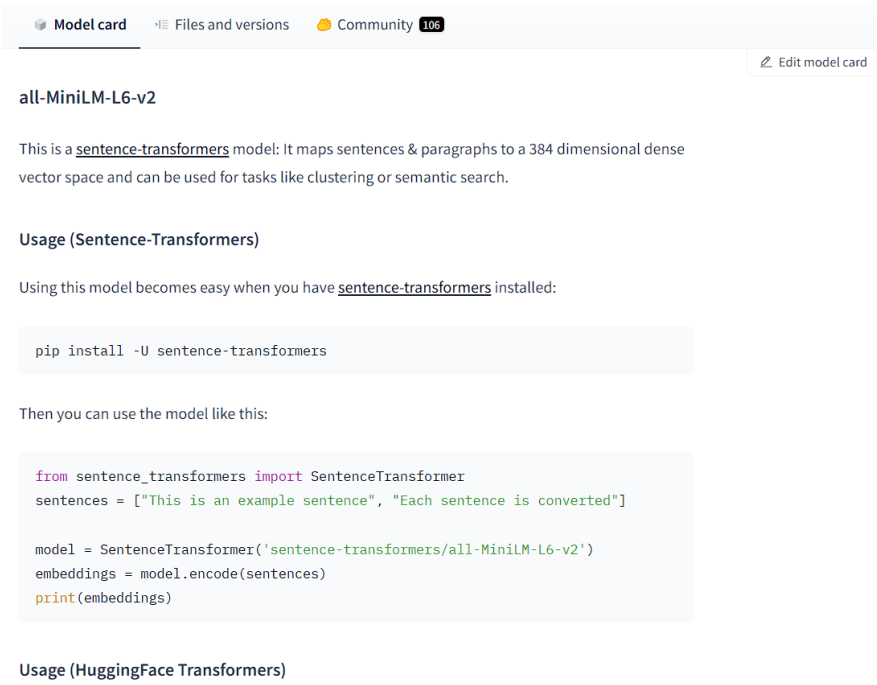

Figura 2.1: Esempio di model card

cessivo spesso di farà riferimento al termine *card* indicando direttamente il codice Python all'interno della documentazione del PTM, dato che l'intero obbietivo si concentra su questo aspetto.

2.2 Processo di raccolta dati

Il *data mining*, in questo contesto si riferisce all'attività svolta durante il percorso di tirocinio. Il processo rappresentato in Figura 2.2 si basa sulla raccolta, filtraggio e strutturazione degli script Python provenienti da repository pubblici su GitHub, al fine di individuare esempi reali di utilizzo dei modelli di Hugging Face. Questa fase è fondamentale per costruire un dataset affidabile e rappresentativo, propedeutico all'analisi dei pattern di codice e il successivo confronto con le model card ufficiali. L'analisi è iniziata dal dataset di HF [1] disponibile sotto licenza open-source. Preprocessando e filtrando per specifici task, si è focalizzata l'attenzione su un sotto-insieme di 17,660 PTM, che rappresenta circa il primo 10% dei dati, ordinati per numero di download da parte della community.

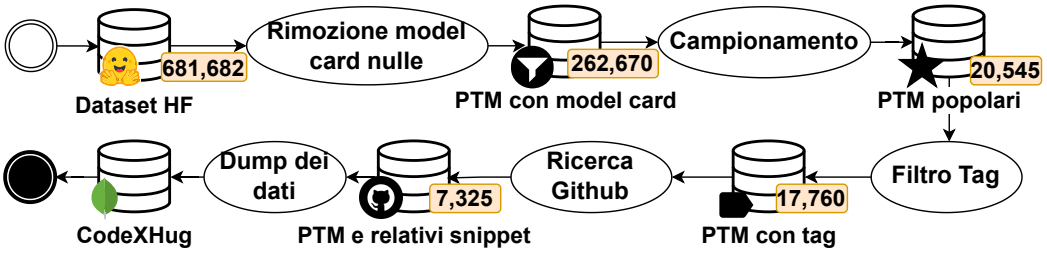

Figura 2.2: Processo di collezione [5]

Per identificare i file Python che utilizzano i modelli, si costruiscono delle specifiche query da inviare all'endpoint dell'API di Github[3], tramite il client PyGithub[4], strutturandole in questo modo: *model_name* language:*Python* La funzione di ricerca, eseguita in parallelo, raccoglie, per ogni modello, al massimo 1,000 (limite imposto dalla Rest API) file content in formato utf-8, archiviandoli su una base dati MongoDB [5].
Partendo da 17,760 query, 10,335 hanno prodotto zero pagine di risultato per un totale cioè di 7,325 modelli di cui si sono collezionati 453,260 script Python suddivisi come in Tabella 2.1 per i task selezionati. L'insieme dei file costituisce il punto di partenza dell'approccio adottato in questo lavoro.

Tabella 2.1: Tags selezionati ed occorrenze dei PTMs

Tag	URLs	Contents
text-generation	181,153	177,913
text-to-image	30,918	30,377
image-classification	37,741	30,377
text-classification	29,977	25,271
text2text-generation	32,743	31,541
fill-mask	68,877	66,575
sentence-similarity	22,473	21,845
question-answering	9,784	9,132
summarization	8,057	7,820
zero-shot-image-classification	14,132	13,900
image-to-text	13,833	13,470
object-detection	7,023	6,858
image-segmentation	12,002	11,725

2.3 LLM

LLM (*L*arge *L*anguage *M*odels) sono modelli di linguaggio che rappresentano un'innovazione significativa nell'ambito del *NLP* (Natural Language Processing), sono costruiti su reti neurali profonde[7], in particolare sull'architettura *Transformer* [26] che introduce un nuovo paradigma per l'elaborazione del linguaggio naturale.
L'architettura Transformer è costituita principalmente da due componenti: l'*encoder*, che elabora il testo in input generando rappresentazioni semantiche, e il *decoder*, che utilizza tali rappresentazioni

[3] `https://docs.github.com/en/rest?apiVersion=2022-11-28`
[4] `https://pygithub.readthedocs.io/en/stable/index.html`
[5] `https://pymongo.readthedocs.io/en/stable/`

per generare l'output. Il processo dell'encoder avviene attraverso una serie di passaggi fondamentali. La prima fase è la tokenizzazione, che converte ogni parola in *token*, ovvero unità fondamentali di elaborazione. Successivamente, ciascun token viene trasformato in un vettore numerico attraverso la fase di embedding, in cui ogni token viene mappato a una rappresentazione numerica basata su un dizionario del modello [6]. Poiché i Transformer non elaborano il testo in maniera sequenziale, si applica il *positional encoding*, che aggiunge un vettore di posizione per preservare l'ordine delle parole [26]. A questo punto entra in gioco il *self-attention layer*, che consente a ciascuna parola di "guardare" tutte le altre nel contesto per comprenderne meglio il significato [26]. Infine, la rappresentazione generata viene ulteriormente elaborata tramite una *feed-forward network (FFN)*, che raffina l'informazione catturando aspetti sintattici e semantici più complessi [26].
Il decoder utilizza le rappresentazioni generate dall'encoder per produrre il testo di output, seguendo un processo altrettanto complesso. Una delle componenti fondamentali di questa fase è il *masked self-attention layer*, che impedisce al modello di analizzare le parole future durante la generazione del testo, assicurando che la predizione successiva si basi esclusivamente sui token già generati. Il *encoder-decoder attention layer* svolge un ruolo cruciale nell'integrare il contesto dell'input con il testo in generazione: esso assegna un punteggio, in una sorta di mappa, a ciascun token dell'input per determinare quali siano i più rilevanti nel processo di generazione. Infine, le informazioni passano attraverso la feed-forward network, che ne raffina ulteriormente la rappresentazione e le trasmette a una funzione *softmax*. Quest'ultima calcola la probabilità di ogni possibile parola successiva e seleziona quella con il valore più alto, completando così il processo di generazione [26]. L'elemento chiave nell'architettura Transformer è il **meccanismo dell'attenzione**, presente sia nell'encoder che nel decoder e permette all'LLM di pesare l'importanza di una parola rispetto alle altre. Il calcolo dell'attenzione è espresso nella seguente equazione[26]:

$$\text{Attenzione}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

dove Q (Query) rappresenta la parola che sta cercando informazioni, K (Key) è la rappresentazione delle parole potenzialmente rilevanti e V (Value) contiene le informazioni effettive associate a ogni parola. Le matrici Q,K,V rappresentano in sostanza tre prospettive diversa di una parola, che vengono calcolate moltiplicando il vettore di embedding per la matrice dei pesi, ovvero i parametri del modello. Infine la softmax, converte i pesi in probabilità, determinando l'importanza relativa di ciascun token.
Nella descrizione delle principali componenti dell'architettura si è parlato indistintamente tra parole e token, ma è fondamentale sottolineare che non sono equivalenti. Un token può corrispondere a un'intera parola, ma in molti casi rappresenta solo una porzione di essa, come prefissi e suffissi, oppure singoli caratteri, specialmente in lingue come il cinese o per simboli speciali. Inoltre, esistono token che rappresentano combinazioni di lettere comuni a più parole, noti come *subword*, che permettono di gestire meglio termini rari e di ottimizzare l'efficienza del modello [15].
Per trasformare le parole del linguaggio naturale in token, ogni LLM utilizza un particolare tokenizer [6], ovvero una componente essenziale che, attraverso un pre-processing, suddivide il testo in unità più piccole e gestibili dal modello. Questo processo include operazioni come rimozione di spazi, normalizzazione e suddivisione in token, garantendo una rappresentazione efficace per la suc-

[6] `https://huggingface.co/docs/transformers/en/main_classes/tokenizer`

cessiva elaborazione. Esistono diverse tecniche di tokenizzazione, una delle principali è la *Subword Tokenization* ovvero una metodologia utilizzata per dividere le parole in parti più piccole, chiamate sotto-unità o subword. Questo aiuta i modelli a gestire sia parole molto comuni che parole rare, senza bisogno di un grande vocabolario. Tra le principali tecniche emerge Byte pair encoding [23] in cui parole comuni (codificate in byte) possono rimanere intere, mentre parole rare vengono suddivise in unità più piccole, consentendo al modello di gestire anche termini sconosciuti attraverso la combinazione di sotto-unità già apprese.
È bene precisare che le componenti appena descritte sono la base delle varie implementazioni più complesse degli LLM che in base al task per cui sono stati progettati impiegano varianti dell'architettura. Per esempio, ad oggi, tra i modelli di linguaggio più popolari spiccano *BERT* [6] che utilizza solo l'encoder per compiti di comprensione e classificazione, *GPT* [20] solo decoder, ideale per generazione di testo, completamento, mentre *T5* (Text-To-Text Transfer Transformer) [21] che servendosi sia dell'encoder che decoder eccelle in traduzione automatica e riassunto.
Gli LLM sono stati addestrati su un'enorme quantità di dati testuali provenienti da libri, articoli e pagine web acquisendo una forte conoscienza e trovando impiego in numerosi ambiti tra cui la generazione e completamento di codice, assistenza nella documentazione software e traduzioni, sintesi di testuali [7], partendo sempre da un input testuale.
Impartire al modello il compito da compiere non è banale per questo si parla di *prompt engineering* come l'arte del formulare input che guidino l'LLM verso le risposte desiderate. Questo può avvenire secondo queste principali tecniche [7]:

- *Few-shot prompting*: inserire nell'input pochi esempi per orientarlo nello svolgimento di un compito, riducendo la necessità di un addestramento approfondito (fine-tuning)
- *Zero-shot prompting* al contrario del precedente, affida al modello un compito senza fornire esempi, sfruttandone la sua capacità di generalizzare.

Nel prompt, per ottenere risposte accurate e contestualizzate in specifici ambiti, solitamente si assegna un ruolo che il modello deve interpretare. Questo aiuta l'LLM a rispondere in maniera più pertinente al contesto richiesto.
Un'importante limitazione di questi modelli di linguaggio sono le *allucinazioni* ovvero un comportamento che tende a generare risposte plausibili ma sbagliate o prive di fondamento logico [7]. Ciò può essere dovuto a comprensione errata del contesto oppure alla difficoltà nell'interpretare input ambigui o complessi. Nel contesto di questo esperimento, un esempio di allucinazione si potrebbe verificare quando l'LLM riceve in input del codice Python e deve estrapolarne pattern d'uso ricorrenti, ma genera funzioni che non hanno alcun collegamento con il codice originale. Questo può essere dovuto all'incapacità del modello di generalizzare correttamente il compito assegnato. Per mitigare questo problema, si è specificato nel prompt cosa fare quando il modello non è in grado di rispondere. Ad esempio, si può esplicitamente indicare che, in caso di incertezza, bisogna restituire una stringa vuota in modo che la successiva analisi dell'output sia più semplice.
Negli ultimi anni, gli LLM open-source hanno acquisito un ruolo sempre più centrale, offrendo un'alternativa accessibile e personalizzabile rispetto ai modelli proprietari. Tra questi spicca **LLaMA** (Large Language Model Meta AI)[25], sviluppato appunto da Meta.
LLaMA è una famiglia di modelli linguistici autoregressivi (ovvero per la generazione del prossimo token si basano su quelli già generati), ottimizzati per garantire prestazioni elevate pur mantenendo

un'efficienza computazionale adeguata. La loro architettura si basa sull'architettura *Decoder-only* ovvero non utilizza l'encoder per generare il testo, prevedendo sotto-sequenze di token man mano che l'output viene costruito [11].
Questi modelli sono disponibili in diverse dimensioni (7B, 13B, 33B, 65B parametri in base alla versione) e possono essere utilizzati sia per applicazioni di completamento del testo, sia per attività più avanzate come la capacità di seguire istruzioni e generare codice [11].
Quando si utilizza un LLM open-source come LLaMA per generare del testo, è possibile personalizzare il comportamento del modello attraverso diversi parametri, che influenzano la creatività, la coerenza e la lunghezza delle risposte. I principali sono:

- *Max tokens*: imposta il numero massimo di token che il modello può generare in una singola risposta. Valori alti consentono generazioni più lunghe, ma possono consumare rapidamente la finestra di contesto (input e prompt di sistema)

- *Temperatura*: controlla il livello (generalmente in un intervallo tra 0.0 - 2) di casualità nella generazione del testo[19]. Valori bassi (es. 0.1 - 0.3) rendono le risposte più deterministiche e precise, con meno variazioni tra chiamate successive, mentre valori alti (es. 0.7 - 1.5) aumentano la creatività e diversità, ma allo stesso tempo possono produrre risposte meno coerenti.

- *Top_p*: filtra i token meno probabili in base a una soglia cumulativa di probabilità. Valori bassi (es. 0.3 - 0.5) rendono i risultati più conservativi, facendo sì che il modello scelga solo tra i token più probabili, mentre valori più alti (es. 0.9 - 1.0) permettono una maggiore diversità nella generazione del testo.

- *Top_k*: limita la scelta del modello ai primi k token più probabili in ogni passo di generazione, dove k è un numero naturale. Un valore basso riduce la casualità, poiché il modello considera solo un insieme ristretto di opzioni, mentre un valore più alto aumenta la varietà delle risposte, ma può introdurre più casualità e incoerenza. Se il parametro non è impostato l'LLM considera tutti i token disponibili in base alle loro probabilità [7].

2.4 Clustering

Il clustering è una tecnica che consiste nell'organizzare oggetti in gruppi, detti cluster, di elementi simili tra loro e distinti dagli altri gruppi, definita anche come *classificazione non supervisionata*, perché non si conoscono a priori né il numero di classi né le loro caratteristiche [3]. Nel campo dell'informatica trova numerose applicazioni nell'ingegneria, manutenzione del software e reverse engineering[3].
Esistono due principali approcci per la creazione di cluster, in base all'algoritmo utilizzato:: *clustering gerarchico* e *clustering partizionale*, ma entrambi i metodi possono essere applicati solo quando è possibile calcolare una misura di prossimità o distanza tra gli elementi, che consente di raggrupparli in base alla similarità o dissomiglianza [3].
In questo lavoro, viene adottato l'approccio partizionale, che che prevede la suddivisione dei dati

[7] `https://rumn.medium.com/setting-top-k-top-p-and-temperature-in-llms-3da3a8f74832`

in un numero fisso di partizioni, ottimizzando una funzione obiettivo basata su una metrica di distanza. In particolare si adopera l'algoritmo *K-means* che segue generalmente il processo in Figura 2.3[8], in particolare il funzionamento si basa nel seguente modo [18]:

- *Inizializzazione dei centroidi*: i centroidi, ovvero i punti che rappresentano il centro di ciascun cluster, vengono inizializzati in modo da essere il più distanti possibile tra loro. Questa fase parte dagli embedding degli elementi di input, ossia la loro rappresentazione numerica.
- *Assegnazione dei punti ai cluster*: ogni embedding viene assegnato al cluster il cui centroide è più vicino in termini di distanza euclidea tra i due. Formalmente la distanza tra un embedding x e un centroide c è calcolata:

$$d(x, c) = \sqrt{\sum_{i=1}^{d}(x_i - c_i)^2} \tag{2.2}$$

  dove, d rappresenta la dimensione del vettore di embedding utilizzato nell'applicazione pratica. nell'applicazione pratica.
- *Ricalcolo dei centroidi*: una volta assegnati tutti i punti, per ogni cluster si calcola un nuovo centroide come media aritmetica (punto o vettore medio) degli embeddings appartenenti al cluster. Se C_j è l'insieme dei punti del cluster j allora il nuovo cluster c_j è definito:

$$c_j = \frac{1}{|C_j|} \sum_{x \in C_j} x \tag{2.3}$$

- *Iterazione fino alla convergenza*: I passaggi di assegnazione e ricalcolo dei centroidi vengono ripetuti fino a quando i centroidi non si stabilizzano ovvero non cambiano significativamente tra due iterazioni consecutive, segnando che l'algoritmo sta convergendo.

La funzione obiettivo del K-Means, che l'algoritmo cerca di minimizzare, è la somma delle distanze quadratiche tra ciascun punto x e il centroide del cluster a cui è assegnato. Formalmente, la funzione obiettivo è definita come:

$$J = \sum_{j=1}^{k} \sum_{x \in C_j} \|x - c_j\|^2 \tag{2.4}$$

dove:

- k è il numero totale di cluster,
- C_j è l'insieme dei punti appartenenti al cluster j,
- c_j è il centroide del cluster j,
- $\|x - c_j\|^2$ è la distanza euclidea al quadrato tra il punto x e il centroide del cluster.

Minimizzando questa funzione, il K-Means cerca di creare gruppi di punti compatti e ben separati, migliorando così la qualità del raggruppamento.

[8] `https://www.ejable.com/tech-corner/ai-machine-learning-and-deep-learning/k-means-clustering/`

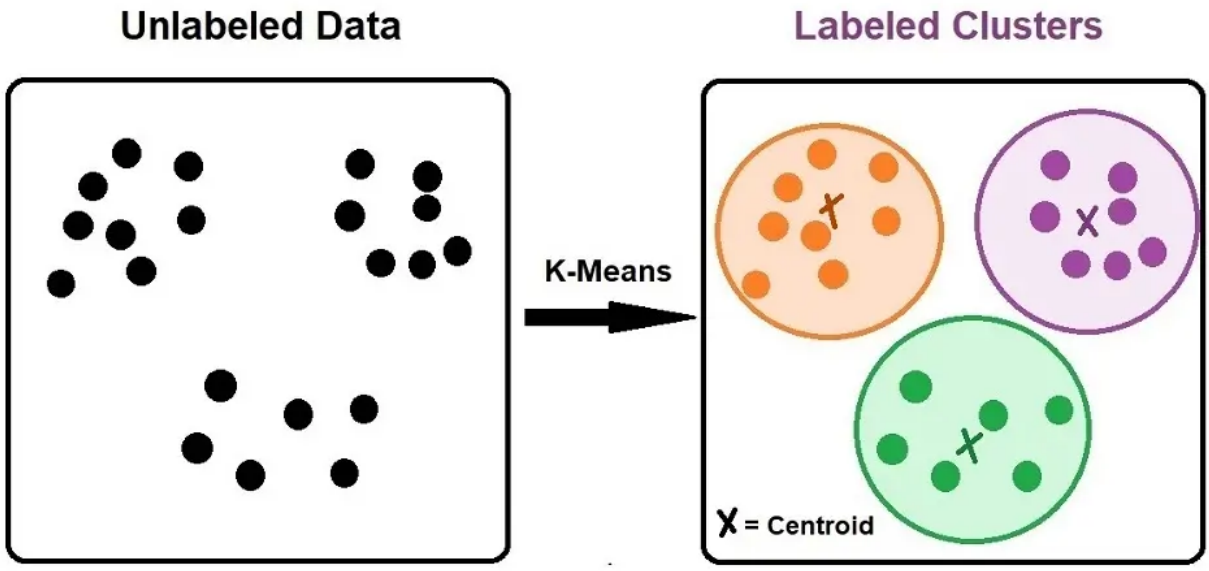

Figura 2.3: Processo di clustering

2.5 Motivazione

Oltre alla misurazione di quanto la generazione automatica di codice, che rappresenti l'uso del modello in contesti reali, sia simile a quella ufficiale presente sulla piattaforma Hugging Face, i risultati condotti dall'esperimento possono essere interpretati anche come una fonte di arricchimento, soprattutto nelle model card in cui non ci sono esempi di codice.
Prendendo come esempio in Figura 2.4, la model card del modello *microsoft/deberta-large-mnli*[9], si può notare che è priva di esempi di codice. Il modello *microsoft/deberta-large*, in generale, è una versione avanzata di BERT e RoBERTa[14], sviluppata da Microsoft, progettata per eccellere nei compiti di comprensione del linguaggio naturale (Natural Language Understanding, NLU) come come analisi del sentiment e classificazione del testo.
La variante *microsoft/deberta-large-mnli* è stata ulteriormente ottimizzata attraverso un processo di fine-tuning sul dataset Multi-Genre Natural Language Inference (MNLI)[29].
Il fatto che nella model card di questo modello non ci siano snippet di codice Python di utilizzo rende ancora più rilevante lo *scopo* proposto in questa tesi. L'assenza di riferimenti pratici all'implementazione del modello conferma la necessità di un approccio automatizzato per l'estrazione e la generazione di esempi esplicativi. Il processo sviluppato in questo lavoro è stato in grado di generare un codice presente nel Listing 2.1 per il PTM in questione, dimostrando la validità dell'approccio nel colmare le lacune presenti nella documentazione ufficiale.

[9] `https://huggingface.co/microsoft/deberta-large-mnli`

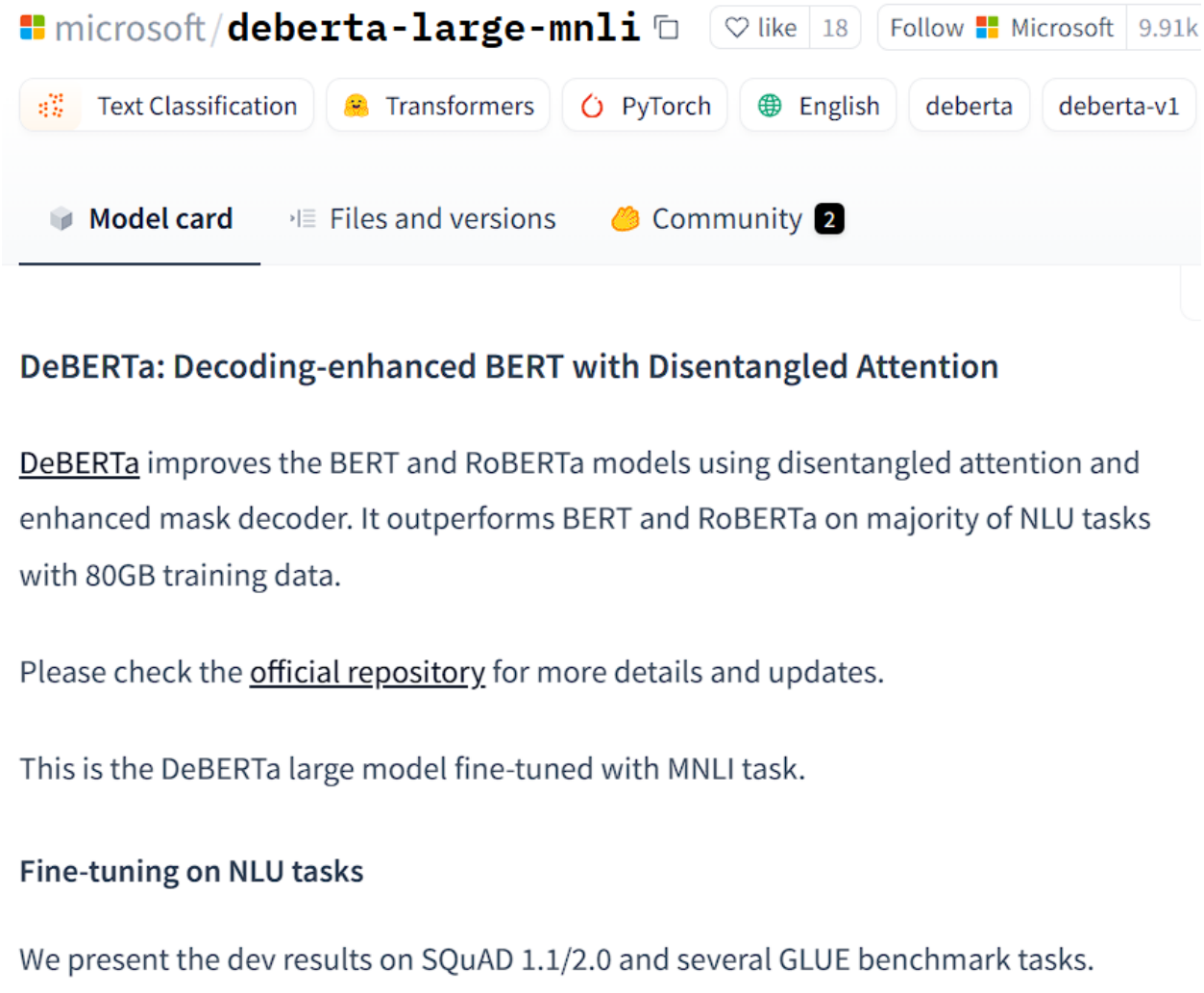

Figura 2.4: Esempio model card senza codice

```
1  def calculate_rewards(self, gt_caption, samples, image):
2      gt_per_sample = [gt_caption[i] for _ in range(len(samples)) for i in range(self.
           num_ref)]
3      samples = [samples[i] for i in range(len(samples)) for _ in range(self.num_ref)]
4
5      with torch.no_grad():
6          toks = self.tokenizer(gt_per_sample, samples, return_tensors='pt', padding=
               True, truncation=True).to(self.device)
7          logits = self.model(**toks).logits
8          probs = logits.softmax(axis=-1)
9          probs = probs.cpu().numpy()
10
11     if self.type == 'nli_p_contradict':
12         rewards = 1 - probs[:,0].reshape(-1, self.num_ref).mean(axis=-1)
13         rewards = (rewards - 0.5) * 2
14
15     return {'bart_nli' : rewards.tolist()}
16
17 class Debertav3NLI(reward_model):
18
19     def __init__(self, config, type='nli_p_contradict'):
20         super().__init__(config)
21
22         if type not in ['nli_p_contradict']:
23             raise '{} type for NLI reward model is not supported'.format(type)
24         else:
25             self.type = type
26
27         print('Loading Deberta-v3 NLI model...')
28         start = time.time()
29         self.tokenizer = AutoTokenizer.from_pretrained('microsoft/deberta-large-mnli
               ', cache_dir=config['cache_dir'])
30         self.model = AutoModelForSequenceClassification.from_pretrained('microsoft/
               deberta-large-mnli', cache_dir=config['cache_dir'])
31         self.model.to(self.device)
32         self.model.eval()
33         end = time.time()
34         print(f'NLI model loaded; {end - start:.2f}s elapsed')
35         self.num_ref = config['num_ref'] # number of reference captions per image
36
37     def forward(self
```

Listing 2.1: Esempio di output della generazione

L'esempio nel Listing 2.1, sebbene non rappresenti un'implementazione completa, fornisce dettagli tecnici utili che possono essere interpretati e adattati da sviluppatori umani per applicazioni specifiche.

In particolare la classe Debertav3NLI (riga 16-33) inizializza il modello e il tokenizer da Hugging Face con le librerie "AutoTokenizer" (riga 29) e "AutoModelForSequenceClassification" (riga 30). Il modello viene poi trasferito sul dispositivo di calcolo (self.device) e impostato in modalità di valutazione (eval()) per disabilitare la fase di addestramento. La funzione calculate_rewards() (riga 1-15) probabilmente è progettata per calcolare un punteggio di ricompensa sulla base della simila-

rità tra una didascalia di riferimento (gt_caption) e un insieme di campioni generati (samples). Il codice utilizza PyTorch[10] per tokenizzare le sequenze di input sfruttando il tokenizer del modello in questione, specificando le opzioni di padding e truncation per garantire la compatibilità con la lunghezza massima supportata. Il risultato viene trasferito sul dispositivo di calcolo (self.device). Infine le righe 7-8 calcolano i punteggi del modello (logits), che vengono convertiti in probabilità tramite una funzione softmax.
Nel complesso, il codice mostra un'implementazione chiara dell'uso di DeBERTa per valutare la qualità di generazioni testuali. Sebbene non sia una pipeline completa, fornisce elementi essenziali, come il caricamento del modello, la tokenizzazione, l'inferenza e la gestione delle probabilità, che possono essere estesi o adattati a scenari più complessi.

[10] `https://pytorch.org/`

Capitolo 3

Lavori correlati

In questa sezione vengono presentati i principali studi e strumenti esistenti che affrontano tematiche affini a quelle trattate in questo esperimento. L'analisi della letteratura permette di contestualizzare il lavoro svolto, evidenziando le soluzioni già proposte, i loro punti di forza e le eventuali limitazioni.

3.1 Sistemi tradizionali per riuso di codice

Il riuso del codice è un aspetto centrale nello sviluppo software, in quanto consente di ridurre il tempo di implementazione, migliorare la qualità del software e facilitarne la mantenibilità. Tuttavia, individuare frammenti di codice riutilizzabili e integrarli in nuovi progetti rappresenta una sfida complessa.
Esempi significativi di sistemi che si collocano in questo contesto sono FOCUS[16], UP-Miner[27], PAM[10] e MAPO[31], che si basano su tecniche di data-mining per estrarre pattern d'uso dal codice open-source.
FOCUS è un sistema di raccomandazione di chiamate API e snippet di codice basato su un approccio di collaborative filtering. Il sistema analizza grandi quantità di codice open-source per identificare pattern di utilizzo delle API e suggerire frammenti contestualizzati di codice pertinenti agli sviluppatori. L'efficacia del sistema è stata dimostrata su un dataset di 2,600 applicazioni Android, evidenziando prestazioni superiori rispetto a metodi esistenti permettendo di superare le limitazioni delle fonti tradizionali, come Stack Overflow[1] o documentazioni ufficiali, che spesso forniscono esempi incompleti o generici.
Anche UP-Miner è un sistema che estrae pattern di utilizzo delle API dai repository software. L'obiettivo è ridurre la ridondanza e garantire che i pattern estratti abbiano un'ampia copertura. Il sistema utilizza un algoritmo di sequence mining con una doppia fase di clustering per raggruppare pattern simili e migliorare la qualità delle raccomandazioni. Applicato su un dataset Microsoft da 8,5 milioni di linee di codice, UP-Miner ha ridotto la percentuale di pattern ridondanti dal 51.12% all'11.92%.
PAM, invece, adotta un approccio diverso, basato su un modello probabilistico per filtrare i pattern API irrilevanti. A differenza di UP-Miner, che si concentra sulla frequenza delle sequenze, PAM stima la probabilità che un pattern sia significativo rispetto al puro caso, riducendo così il rumore nei suggerimenti. Applicato a un dataset GitHub da 54,911 metodi client e 4 milioni di linee di

[1] `https://stackoverflow.com/questions`

codice, PAM ha ottenuto una precisione del 69% nel recupero di pattern API rilevanti.
Infine, MAPO è un sistema progettato per aiutare gli sviluppatori a comprendere e riutilizzare le API estraendo pattern d'uso direttamente da repository open-source. Il problema è che gli strumenti di ricerca del codice, come Google Code Search[2], restituiscono spesso troppi snippet non organizzati, rendendo difficile individuare quelli realmente utili. MAPO propone una soluzione basata su mining di pattern API e clustering, per raggruppare snippet di codice con usi simili e presentarli agli sviluppatori in modo strutturato. In particolare, il clustering viene utilizzato per organizzare i frammenti di codice prima dell'estrazione dei pattern (con frequent subsequence mining), migliorando la qualità delle raccomandazioni. Il sistema identifica le sequenze di chiamate API presenti nei metodi analizzati e le raggruppa in cluster distinti, basandosi su tre criteri principali: somiglianza nei nomi dei metodi, somiglianza nei nomi delle classi e somiglianza nelle API chiamate.
Il lavoro presentato in questa tesi si allinea a questi approcci tradizionali, condividendo l'obiettivo di estrarre pattern d'uso dal codice open-source per aiutare i programmatori nello sviluppo software, riducendo gli sforzi manuali. Tuttavia, mentre i sistemi come FOCUS, UP-Miner, PAM e MAPO si concentrano sull'estrazione di pattern d'uso delle API, questa ricerca si distingue per l'analisi dell'effettivo utilizzo dei modelli di Hugging Face, il cui accesso avviene principalmente tramite API, con l'obiettivo di migliorare la documentazione e facilitare l'integrazione dei modelli negli ambienti di sviluppo reali.

3.2 Generazione di codice tramite LLM

L'uso dei modelli di linguaggio di grandi dimensioni ha rivoluzionato il modo in cui gli sviluppatori scrivono software, automatizzando diverse attività di programmazione. Lo studio[11] presenta un'analisi sistematica della letteratura scientifica applicata all'ingegneria dell'software, analizzando oltre 395 articoli pubblicati tra il 2017 e il 2024, con l'obbiettivo di classificare i principali approcci adottati nell'applicazione dei LLMs al software engineering.
L'articolo suddivide l'analisi in tre categorie di modelli di linguaggio: encoder-only, encoder-decoder e decoder-only, ognuno con caratteristiche architetturali diverse specifiche per diverse attività di programmazione.
I modelli encoder-only, come BERT e CodeBERT[9] vengono impiegati principlamente per attività di analisi del codice tra cui il rilevamento di bug e la sintesi del codice in linguaggio naturale. Gli encoder-decoder, tra cui T5 e CodeT5[28], si rivelano particolarmente efficaci nella traduzione tra linguaggi di programmazione e nella generazione di documentazione automatizzata. Infine, i decoder-only, come Codex[4] e GPT-4 sono ottimizzati per la generazione e il completamento del codice, risultando i più adatti per applicazioni come GitHub Copilot[3].
Inoltre, si evidenziano anche le principali tecniche adottate per la generazione di codice. Un primo approccio è il prompting basato su istruzioni, in cui i modelli generano codice a partire da una descrizione in linguaggio naturale. Questo metodo è alla base di strumenti come Codex, che permette di ottenere funzioni complete semplicemente descrivendone il comportamento desiderato. Un'altra strategia diffusa è il few-shot e zero-shot learning, che consente ai modelli di generare codice con o senza esempi pregressi, migliorando la capacità di adattamento a compiti nuovi. Infine, il fine-tuning

[2] `https://developers.google.com/code-search`
[3] `https://github.com/features/copilot`

su dataset specializzati permette di ottimizzare i modelli per la comprensione e la generazione di codice di alta qualità.
Un aspetto critico della generazione automatica di codice riguarda la sua valutazione. Gli LLMs vengono tipicamente misurati attraverso metriche come CodeBLEU, l'affidabilità del codice generato rimane una sfida aperta. Il codice prodotto dai LLMs può contenere errori logici o vulnerabilità di sicurezza, e spesso manca di una comprensione profonda del contesto applicativo.
Lo studio effettuato nell'articolo si inserisce direttamente in questo contesto, perché si sfrutta un LLM per sintetizzare pattern di codice rappresentativi basandosi sugli snippet estratti da progetti open-source, un approccio simile a Codex e AlphaCode[12], ma con l'obbietivo di automatizzare e migliorare la documentazione dei PTM di Hugging Face.
L'uso di metriche di similarità come CodeBLEU consente di valutare l'accuratezza dei pattern generati, garantendo che essi non siano una semplice riproduzion degli esempi ufficiali, ma piuttosto una sintesi alternativa.

Capitolo 4

Approccio

Nella Figura 4.1 è raffigurata la pipeline del lavoro partendo dal dataset prodotto nella sezione 2.2, fino alla generazione di pattern riccorenti tramite LLM. La legenda indica, per ogni numero presente, il tipo di dato al quale ci si sta riferendo.

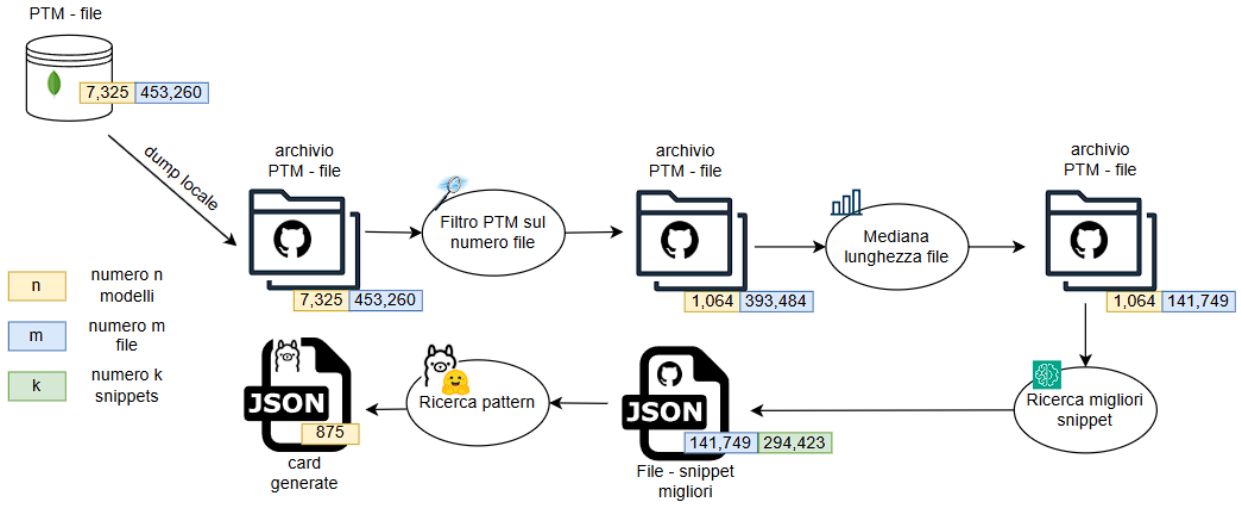

Figura 4.1: Pipeline di lavoro

4.1 Filtro modelli popolari

Il database prodotto nella parte di tirocinio contiene per 7,325 PTM un totale di 453,260 file sorgenti Python. La distribuzione in Figura 4.3 dei dati è fortemente asimmetrica, presentando un'elevata concentrazione di modelli con un numero molto basso di codici mentre solo un ristretto numero di PTM è utilizzato in una quantità significatamente maggiore di file.
In particolare, la distribuzione segue un andamento tipico di quelle *long tail* che si manifesta con una forte asimmetria positiva (right-skewed distribution) evidenziando che la maggior parte dei modelli è utilizzata in pochissimi file mentre modelli altamente popolari sono in una frequenza molto minore.

Infatti dalle informazioni statiche della Tabella 4.1, la mediana è molto inferiore alla media e la deviazione standard elevata conferma l'alta variabilità tra i modelli. Inoltre, la notevole skewness evidenzia che la distribuzione è sbilanciata a destra, con alcuni modelli molto più utilizzati rispetto

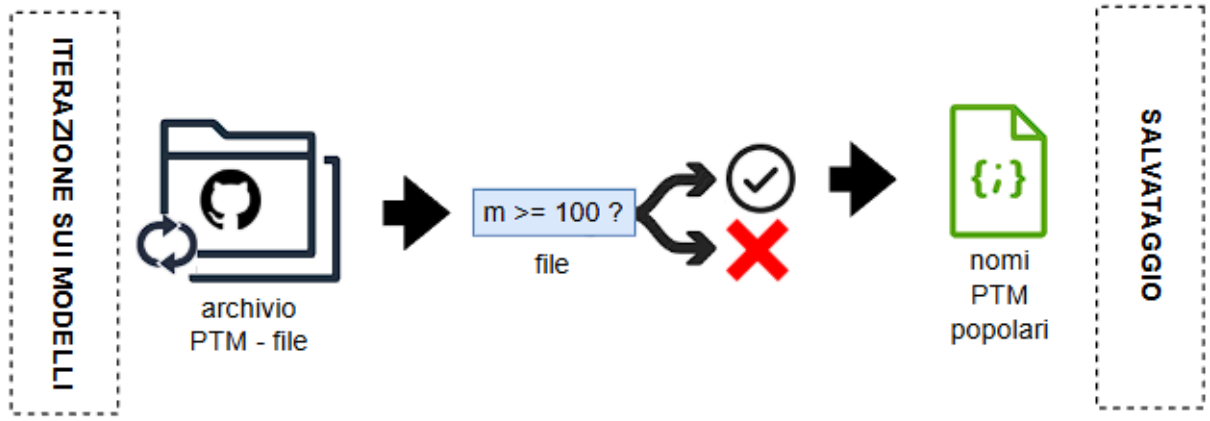

Figura 4.2: Filtraggio dei modelli in base al numero di file

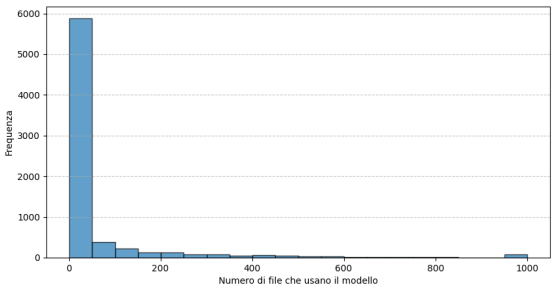

Figura 4.3: Distribuzione dei file

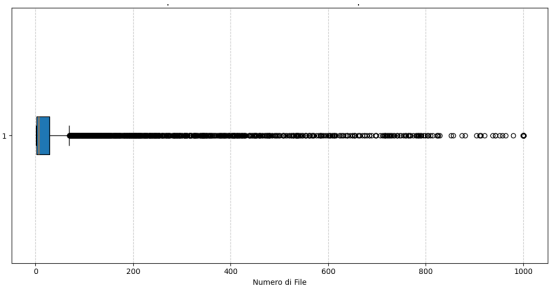

Figura 4.4: Analisi outlier

alla maggioranza. Infine la curtosi indica la presenza di numerosi *outlier* mostrati nella Figura 4.4, cioè modelli con un utilizzo estremamente superiore rispetto alla norma.
Per evitare possibili distorsioni negli esperimenti successivi dell'approccio, si è deciso quindi di selezionare un sottoinsieme di PTM aventi un numero di file arbitrariamente maggiore di 100, formato da 1,064 modelli per un totale complessivo di 393,484 script.
Dopo aver applicato il filtro selezionando solo i modelli con almeno 100 file associati, la distribuzione

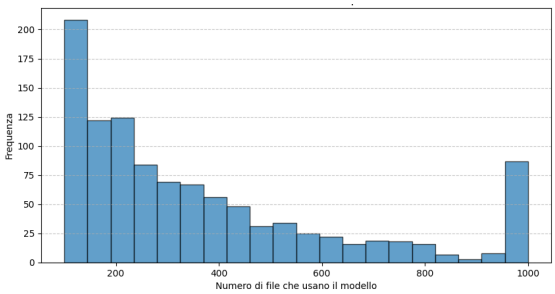

Figura 4.5: Distribuzione dei file filtrati

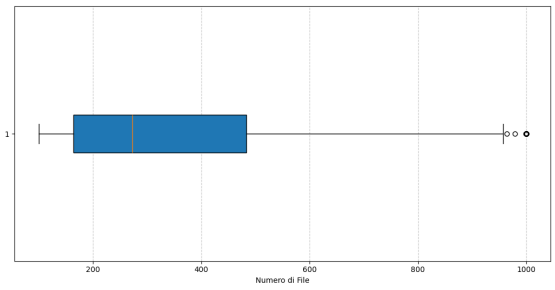

Figura 4.6: Analisi outlier post filtro

nella Figura 4.5 mostra caratteristiche significativamente diverse rispetto a quella originale.
Inoltre, dalle nuove metriche nella Tabella 4.2 la media è aumentata da 63.99 a 369.82, e la mediana è passata da 6 a 272.50, indicando che ora ci si sta concentrando solo sui modelli più popolari. La deviazione standard di 265.65 mostra ancora una notevole variabilità nel numero di file per modello, sebbene ridotta rispetto alla distribuzione completa. L'asimmetria (skewness) si è ridotta drasticamente da 3.86 a 1.17, suggerendo che la distribuzione è ora meno sbilanciata a destra.

Metrica	Valore
Media	63.99
Mediana	6.00
Deviazione Standard	162.54
Skewness	3.86
Kurtosis	16.09

Tabella 4.1: Metriche statistiche per il numero di file per modello

Metrica	Valore
Media	369.82
Mediana	272.50
Deviazione Standard	265.65
Skewness	1.17
Kurtosis	0.31

Tabella 4.2: Metriche per il numero di file per modello post filtro

Inoltre, la curtosi è passata da 16.09 a 0.31, indicando una distribuzione più piatta con meno outlier estremi, come mostrato nella Figura 4.6.
Questo filtraggio ha quindi permesso di eliminare la gran parte dei modelli con utilizzo marginale, garantendo un'analisi più bilanciata dei modelli più diffusi.
Oltre a ridurre le distorsioni dovute alla scarsità di dati, la selezione si è basata anche sul cercare di includere i modelli più popolari e ampiamente utilizzati sulla piattaforma, al fine di concentrarsi sui modelli effettivamente riliventi per la community.
I nomi dei modelli selezionati sono stati salvati in un file JSON in modo da essere facilmente accessibili e utilizzabili nelle fasi successive.

4.2 Filtro dei codici sorgenti per lunghezza

Per migliorare l'efficienza della successiva ricerca delle porzioni più rilevanti dei file si è implementato un processo di selezione degli script mostrato nella Figura 4.7, precedentemente filtrati, basato sulla lunghezza degli stessi.
L'idea di base è che alcuni codici troppo brevi potrebbero non contenere informazioni significative sull'utilizzo di un particolare modello e allo stesso modo codici troppo lunghi potrebbero includere porzioni non pertinenti come ad esempio descrizioni e lunghi commenti.
Il processo di filtraggio si basa nel considerare solo gli script che hanno una lunghezza in termini di linee di codice comprese nell'intervallo interquantile (IQR), utilizzando la mediana come valore centrale. In particolare, l'IQR è una misura statistica della dispersione dei dati così definita:

$$IQR = Q3 - Q1 \tag{4.1}$$

dove Q1 (primo quartile) indica il valore sotto il quale si trova il 25% dei file più corti, mentre Q3 (terzo quartile) il valore sotto il quale si trova il 75% dei file. Quindi, il risultato rappresenta l'ampiezza della fascia centrale della distribuzione cioè pari al 50% dei dati.

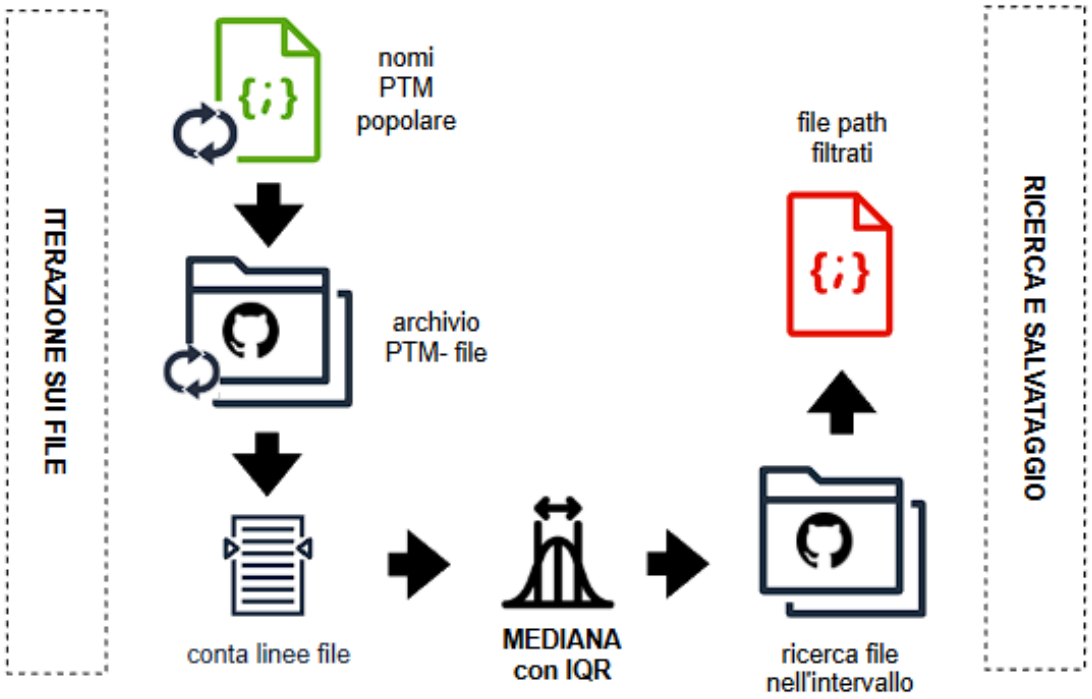

Figura 4.7: Filtraggio dei file per intervallo sulla lunghezza

Nella pratica la mediana e l'IQR vengono così impiegati nel calcolo nell'intervallo di selezione:

$$min\_length = max(mediana - 0.25 * IQR, 1) \tag{4.2}$$

$$max\_length = mediana + 0.25 * IQR \tag{4.3}$$

dove il fatore 0.25 permette di restringere l'intervallo intorno alla mediana senza essere troppo rigido. Di conseguenza, questo range permette di selezione i file in base alla loro lunghezza attraverso una misura più robusta della media, concentrandosi su insieme più omogeneoe ed escludendo la presenza di eventuali outlier.
Per ottimizzare il tempo di elaborazione dei vari calcoli, il processo viene eseguito in parallelo utilizzando il multithreading, perché si tratta prevalentemente di operazioni di lettura e scrittura su disco (I/O bound), in quanto calcolo matematico è piuttosto semplice. Nel dettaglio, ogni thread lavora su un sottoinsieme di file, riducendo l'attesa rispetto ad un'elaborazione sequenziale. Inoltre, l'uso di un *ThreadPoolExecutor*[1] garantisce una gestione automatica delle risorse, bilanciando il carico di lavoro tra i thread attivi.
I path dei file la cui lunghezza rientra nell'intervallo dinamico calcolato, vengono salvati in un file JSON cosicchè da avere un riferimento diretto all'archivio locale del dataset.

4.3 Ricerca dei migliori code snippet

L'LLM preso in considerazione in questo esperimento ha una finestra di contesto molto limitata, ovvero non riesce a gestire per ogni chiamata più di un certo numero di token complessivamente in

[1] `https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor`

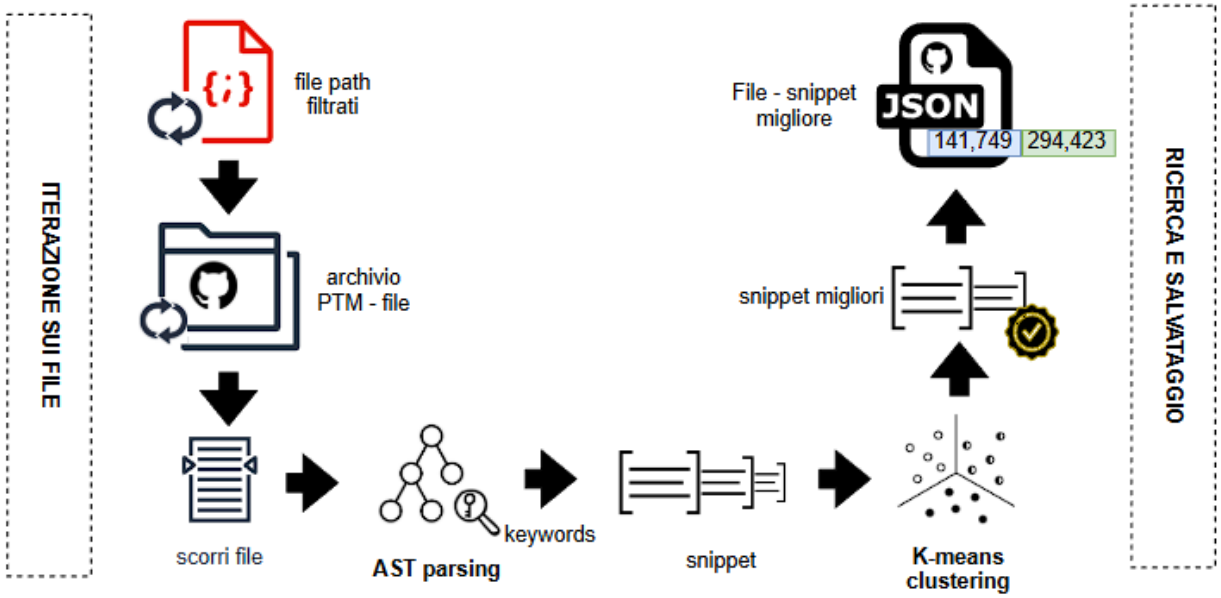

Figura 4.8: Processo di ricerca degli snippet di codice

input e in output.
L'ipotetico approccio di far valutare al modello direttamente interi script Python, senza un pre-processing, potrebbe essere molto restrittivo in termini di possibili pattern di codice da scoprire, nonché inefficiente perché gli stessi script potrebbero non avere un interessante senso semantico. Per esempio, alcuni file, si potrebbere limitare soltanto a menzionare il modello in esame con commenti e descrizioni senza poi impiegarlo realmente.
Per affrontare queste problematiche, si è deciso di analizzare in maniera automatica ogni singolo file filtrato dalla sezione precedente, con l'obiettivo di estrapolarne delle porzioni che fossero le più rappresentative e rilevanti possibile in termini di utilizzo del modello. Quindi, provando a non considerare per esempio sezioni di debugging o funzioni di utility non significative. Il processo di estrazione mostrato nella Figura 4.8 è basato sulla ricerca di parole chiavi legate fortemente all'addestramento, tokenizzazione e inferenza standard dei PTM disponibili su HF evidenziate per intere nel Listing 4.1. In particolare, l'implementazione è suddivisa in 5 fasi:

4.3.1 Analisi della struttura del codice

Per ogni file Python, risultante dal calcolo della mediana, viene utilizzo un parser per l'*Abstract Syntax Tree (AST)*[2], ovvero una rappresentazione sintattica strutturata del codice sorgente. Ciò permettere di iterare sui nodi dell'albero per identificare importazioni di librerie e definizioni di funzioni (incluso il loro corpo) che contengono (match) almeno una delle keyword definite.
La ricerca della corrispondenza viene effettuata a livello di linea, quindi per fornire un contesto più ampio e facilitare la comprensione dello snippet estratto, vengono aggiunte due (arbitrariamente) righe di contesto sia prima che dopo la porzione individuata.
Tuttavia, l'analisi AST potrebbe fallire in presenza di errori sintattici nel codice sorgente come parentesi mancanti, indentazioni errate, stringhe non chiuse correttamente. In questi casi l'errore

[2] `https://docs.python.org/3/library/ast.html`

viene solo segnalato a schermo e il file scartato dal processo successivo.

```
import, from, transformers, huggingface,
AutoModel, AutoTokenizer, AutoConfig, pipeline,
train, fit, fine_tune, finetune,
token, tokenizer, encode, decode,
predict, inference, generate, metric,
sklearn, scikit-learn
```

Listing 4.1: parole chiavi da ricercare nel codice

4.3.2 Raggruppamento delle informazioni

Successivamente all'individuazione delle diverse porzioni pertinenti può accadere che più frammentidi siano vicini tra loro condividendo alcune righe di codice. Per evitare ripetizioni ridondanti, questi frammenti vengono unificati mediante la fusione degli intervalli numerici di righe sovrapposti, garantendo che snippet contigui non vengano considerati separatamente.

4.3.3 Ricerca dei migliori snippet

L'analisi sintattica e il successivo raggruppamento restituiscono, per ogni file, un insieme di snippet contenenti almeno una corrispondenza con le keyword definite. Tuttavia, per scegliere il migliore tra i candidati è necessario considerare anche il significato semantico del codice, analogamente all'approccio del lavoro[31], che utilizza il clustering per raggruppare snippet di codice con usi simili delle API, migliorando la qualità delle raccomandazioni.
Per raggiungere questo obiettivo, si è utilizzato l'algoritmo *K-means* implementato con la libreria *scikit-learn*[18] che opera su una rappresentanzione numerica delle informazioni. Poiché gli snippet sono inizialmente sotto forma di stringhe, è stato adottato un modello di embeddings[3]https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2 messo a disposizione direttamente da Hugging Face tramite la libreria *sentence_ transformers*[4]. Questo modello, basato sull'architettura *Transformer* e in particolare in una versione ridotta di *BERT*, converte ogni snippet in un vettore numerico di 384 dimensioni attraverso una pipeline di tokenizzazione, calcolo dell'attenzione e infine generazione dell'embedding tramite mean pooling (media pesata di tutti gli embedding dei token). Nonostante i modelli della libreria *sentence_ transformers* siano principalmente progettati per il linguaggio naturale riescono comunque a catturare molte informazioni sintattiche e semantiche nei frammenti di codice soprattutto se l'obbiettivo è selezionare gli snippet rappresentativi e non necessariamente eseguire un'analisi profonda delle operazioni a basso livello di codice. Inoltre, è particolarmente utile quando sono presenti commenti in linguaggio naturale che spiegano determinare strutture e funzioni nel codice.
Si è deciso di adottare questo tipo di embedder anche per un trade-off tra prestazioni e leggerezza, perché utilizzando ad esempio *CodeBERT* [9], i tempi di inferenza e utilizzo della memoria per centinaia di migliaia di file sarebbero stati sicuramente più alti. Gli embeddings ottenuti vengono passati al *K-means* che ha il compito di assegnarli in k cluster. Il valore di k viene scelto empiricamente come il minimo tra un valore fisso pari a 3 e la lunghezza della lista degli snippet, in modo

[3] `\unskip\protect\penalty\@M\vrulewidth\z@height\z@depth\dp`
[4] `https://sbert.net/`

da adattarsi dinamicamente al numero di candidati disponibili.
Una volta che l'algoritmo ha raggiunto la convergenza, per ogni cluster si seleziona lo snippet il cui embedding è più vicino al rispettivo centroide. Tale snippet viene considerato il più rappresentativo del gruppo, poiché riassume al meglio le caratteristiche semantiche condivise dagli altri elementi del cluster.

4.3.4 Elaborazione parallela

Per ottimizzare i tempi di esecuzione, in presenza di un elevato numero di file da elaborare, anche in questa fase, il processo è stato parallelizzato tramite l'utilizzo del *multithreading*. Ciascun thread si occupa di elaborare in modo indipendente un file, estraendo gli snippet, calcolando gli embeddings e applicando l'algoritmo di clustering. Questo approccio consente di ridurre significatamene i tempi complessivi di esecuzione.

4.3.5 Salvataggio informazioni

L'insieme degli snippet selezionati da ogni cluster vengono infine salvati in un file JSON per facilitarne la consultazione e l'utilizzo nella successiva fase di analisi, secondo la struttura presente nel Listing 4.2.

```
{
  "bert-base-uncased": {
    "path/to/file1.py": [
      "import transformers\n\nmodel = AutoModel.from_pretrained('bert-base
         -uncased')",
      "tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')\
         ninputs = tokenizer('Hello world', return_tensors='pt')"
    ],
    "path/to/file2.py": [
      "pipeline = transformers.pipeline('sentiment-analysis')\nresult =
         pipeline('I love coding!')"
    ]
  },
  "roberta-base": {
    "path/to/file3.py": [
      "import transformers\nmodel = AutoModel.from_pretrained('roberta-
         base')"
    ]
  }
}
```

Listing 4.2: Esempio di struttura JSON dei migliori snippet selezionati

Questa pipeline consente di selezionare automaticamente, per ciascun file, le porzioni di codice più rappresentative in termini di utilizzo dei modelli di Hugging Face, riducendo la quantità di codice irrilevante da fornire successivamente all'LLM.

4.4 Utilizzo dell'LLM

Nella Figura 4.9 viene rappresentato il processo di analisi degli snippet trovati nella fase di clustering, al fine di costruire le richieste che il modello in remoto dovrà elaborare. Nel contesto di questo

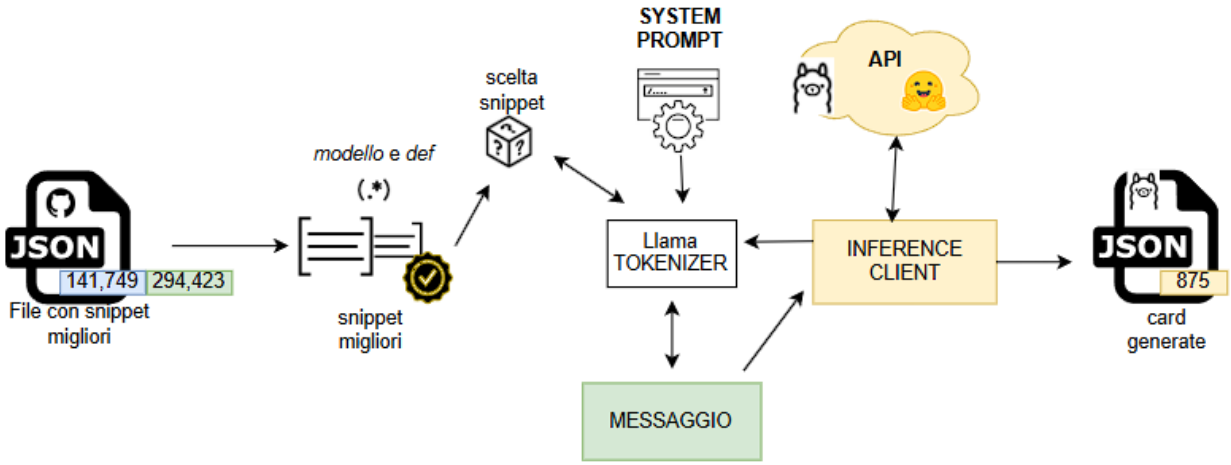

Figura 4.9: Estrazione pattern con LLM

esperimento, è stato sfruttuano un particolare LLM opensource per analizzare gli snippet di codice Python estratti precedentemente, al fine di individuare schemi comuni nell'utilizzo dei PTM tramite le principali librerie della piattaforma di Hugging Face.
L'obiettivo è di isolare pattern significativi senza dover eseguire un'analisi manuale per ciascun modello esaminato, riducendo tempi e sforzi operativi.

4.4.1 Flusso operativo

L'idea di base è fornire, per ogni modello in esame, il maggior numero possibile di snippet, cercando di massimizzare l'utilizzo dei token di input disponibili per l'LLM. Per garantire coerenza e confrontabilità tra le varie chiamate, viene utilizzato un *prompt* fisso per tutti i modelli.
Il prompt ha come unico scopo di istruire il modello a individuare eclusivamente pattern di codice, come:

- importazioni comuni
- definizioni di funzione
- strutture tipiche di utilizzo dei modelli PTM

La tecnica utilizzata nella costruzione è *zero-shot prompting*, perché il modello non riceve esempi espliciti di output atteso all'interno della stringa di input, basandosi interamente sulla capacità del modello di generalizzare nelle risposte.

```
system_prompt = (
"You are an AI assistant specialized in Python.
I will provide you with a series of code snippets extracted from various Python files.
Note that these snippets might not form a coherent or complete code module when combined together.
Your task is to analyze these snippets and extract recurring code patterns, such as function definitions (using 'def'), common imports, and other typical structures found in Python code.
Focus solely on identifying and returning the common patterns in the context of the model {snippet_model}.
Return ONLY the code snippets.
DO NOT include only the common imports.
Do NOT include any explanations, descriptions, or metadata.
Do NOT generate any summaries, bullet points, markdown formatting, or additional text.
If you are unable to find any relevant patterns, please return an empty string."
)
```

Listing 4.3: Prompt per LLama

Si può notare come nel prompt, descritto nel Listing 4.3, venga specificato un ruolo di assistente Python per cercare di contestualizzare al meglio le risposte. Inoltre, si definiscono dei vincoli nella generazione dell'output per evitare che contenga testo potenzialmente non necessario. Queste ipotetiche descrizioni superflue potrebbero alterare la fase di validazione, perché si confronterebbe del linguaggio naturale con codice Python alterando notevolmente i risultati delle metriche.
Per la scelta delle porzioni di codice da inviare al modello, viene effettuato un primo filtro: tramite espressioni regolari vengono selezionati solo gli snippet contenenti almeno il nome del modello e la keyword "def". Questo controllo garantisce che gli snippet siano pertinenti con il modello in esame e che venga inclusa almeno una definizione di funzione, rendendo il codice rilevante per l'identificazione di pattern funzionali.
Una volta selezionati gli snippet l'input per l'LLM viene costruito in maniera incrementale:

- Gli snippet vengono aggiunti uno alla volta alla stringa da inviare
- Dopo ogni aggiunta, si calcola il numero di token complessivi utilizzando il tokenizer del modello
- Il processo continua finché non si raggiunge il limite massimo di token previsto

Il numero massimo di token disponibili per l'input è calcolato come segue:

$$allowed\_input\_tokens = max\_total\_tokens - system\_tokens - reserved\_output\_tokens \quad (4.4)$$

dove

- $allowed\_input\_tokens$ indica numero di token disponibili, che aumenta con il crescere degli snippet selezionati
- max_total_tokens è il limite imposto dal modello pari a 4096 token

- *system_tokens* riguarda il numero token occupati dal prompt di sistema, considerabile costante a eccezione della variazione dovuta al nome del modello
- *reserved_output_tokens* termine costante pari a 400, riservati per l'output dell'LLM

Questo approccio garantisce l'utilizzo ottimale della finestra di contesto del modello, permettendo di massimizzare le informazioni fornite, rispettando i limiti imposti.
Per ottenere risposte più accurate, coerenti e meno soggette a variazioni casuali, nella chiamata all'LLM viene impostata una *temperatura* pari a 0.2. L'obiettivo è l'individuazione precisa di pattern di codice piuttosto che una generazione creativa, quindi l'uso di una bassa temperatura riduce la variabilità tra le risposte e minimizza il rischio di ottenere output fuorvianti o allucinazioni. Inoltre il parametro*top_p* è impostato a 1.0, il che significa che la selezione dei token successivi avviene considerando l'intera distribuzione di probabilità prodotta dal modello senza introdurre un bias artificiale, che potrebbe penalizzare alternative comunque valide. A differenza di valori più bassi di top_p, che avrebbero ristretto la scelta ai token con le probabilità più alte fino a raggiungere la soglia cumulativa, la scelta fatta consente di sfruttare appieno il potenziale del modello senza escludere a priori opzioni meno probabili che potrebbero comunque risultare rilevanti.
La richiesta all'LLM avviene tramite la chiamata nel Listing 4.4.

```
1         completion = client.chat.completions.create(
2             model=llm_model,
3             messages=messages,
4             max_tokens=reserved_output_tokens,
5             temperature=0.2,
6             top_p=1.0
7         )
```

Listing 4.4: Chiamata API

Il parametro *messages* nel Listing 4.4 contiene il contesto della conversazione ovvero il prompt, precendemente definito,associato al ruolo di sistema che l'LLM deve assumere, seguito dal contenuto vero e proprio costruito iterativamente dall'insieme di snippet di codice.
Per ogni risposta ricevuta dall'LLM, l'algoritmo implementa una semplice logica di validazione per verificarne la pertinenza e la qualità. In particolare, il sistema verifica che l'output soddisfi due criteri fondamentali i) che contenga il nome del modello analizzato, confermando che il contesto richiesto sia stato effettivamente considerato e ii) che includa almeno le keyword "import" e "def", per garantire la presenza di sezioni di codice Python, essenziali nell'analisi dei pattern.

Se l'output non soddisfa uno o entrambi questi criteri, il sistema effettua automaticamente fino a due nuovi tentativi con lo stesso input, ripetendo la validazione dopo ogni risposta. Ripetere la chiamata permette di provare a moderare la variabilità nelle risposte dell'LLM, aumentando la probabilità di ottenere un output valido e pertinente.
Nel caso in cui, anche dopo tre tentativi complessivi, la risposta non risulti conforme ai criteri stabiliti, viene comunque considerato valido l'ultimo output generato. Questa scelta evita cicli infiniti e garantisce che almeno un risultato venga prodotto per la successiva fase di testing.
Può comunque accadere che l'output dell'LLM sia vuoto, perché se la fase di scelta degli snippet da inviare non produce corrispondenza, l'intero input è vuoto e quindi, di conseguenza il modello risponde con una stringa nulla.
Infine, L'output viene salvato in un file JSON avente la struttura espressa nel Listing 4.5.

```
{
    "nome_modello_1": {
        "system_prompt": "Testo del prompt contenente il nome del modello
           1",
        "user_input": [
            "Snippet di codice selezionato casualmente 1",
            "Snippet di codice selezionato casualmente 2"
        ],
        "llm_output": "Risultato generato dall'LLM in risposta all'input
           fornito",
        "input_tokens": 1234,
        "reserved_output_tokens": 400,
        "attempts": 1,
        "temperature": 0.2
    },
    "nome_modello_2": {
        "system_prompt": "Testo del prompt Testo del prompt contenente il
           nome del modello 2",
        "user_input": [
            "Snippet di codice selezionato casualmente 1",
            "Snippet di codice selezionato casualmente 2"
        ],
        "llm_output": "Risultato generato dall'LLM in risposta all'input
           fornito",
        "input_tokens": 1456,
        "reserved_output_tokens": 400,
        "attempts": 2,
        "temperature": 0.2
    }
}
```

Listing 4.5: Esempio di struttura JSON nella risposta dell'LLM

In sintesi, questo approccio consente di massimizzare l'efficacia delle chiamate all'LLM, garantendo che:

- gli snippet inviati siano pertinenti e rappresentativi
- le risposte contengano informazioni utili e coerenti con l'obiettivo dell'esperimento
- venga rispettato il limite di token imposto dal modello

4.4.2 Esempio di card generata

Si mostra un esempio di codice generato dall'LLM nel Listing 4.6, in riferimento al modello *kakaobrain/karlo-v1-alpha-image-variations* che si applica generalmente per generare immagini partendo da testo in linguaggio naturale [5].

[5] `https://huggingface.co/kakaobrain/karlo-v1-alpha-image-variations`

```
1  def load_data():
2      # Load data from kakaobrain/karlo-v1-alpha-image-variations
3      data = pd.read_csv('data/train.csv')
4      return data
5
6  def preprocess_image(image):
7      # Resize and normalize image
8      image = tf.image.resize(image, (224, 224))
9      image = tf.image.per_image_standardization(image)
10     return image
11
12 def create_model(input_shape):
13     # Create Keras model
14     model = tf.keras.models.Sequential([
15         tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=
               input_shape),
16         tf.keras.layers.MaxPooling2D((2, 2)),
17         tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
18         tf.keras.layers.MaxPooling2D((2, 2)),
19         tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
20         tf.keras.layers.Flatten(),
21         tf.keras.layers.Dense(128, activation='relu'),
22         tf.keras.layers.Dense(10, activation='softmax')
23     ])
24     return model
25
26 def train_model(model, data):
27     # Train model
28     model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics
           =['accuracy'])
29     model.fit(data, epochs=10)
30     return model
31
32 def evaluate_model(model, data):
33     # Evaluate model
34     loss, accuracy = model.evaluate(data)
35     return loss, accuracy
36
37 def save_model(model):
38     # Save model
39     model.save('model.h5')
```

Listing 4.6: Esempio di output della generazione

Si può notare che, sebbene il codice generato non rappresenti un'implementazione completa del modello *kakaobrain/karlo-v1-alpha-image-variations*, esso fornisce comunque una struttura chiara e coerente con l'utilizzo tipico del modello per la generazione di immagini a partire da testo. In particolare, emergono alcuni aspetti tecnici utili, come il caricamento dei dei dati di addestramento (righe 1-4) probabilmente coppie di testo-immagini, la pre-elaborazione delle immagini che comprende il resize e la normalizzazione (righe 6-10) e infine la definizione dell'architettura della rete neurale (righe 12-24) mediante la libreria Keras[6].

[6] `https://keras.io/`

Questi dettagli, anche se non garantiscono un'implementazione esatta del modello, risultano comunque utili per comprendere l'approccio adottato nella documentazione della model card e possono servire come base per sviluppatori interessati a integrare il modello in un progetto pratico.

Capitolo 5

Validazione e Risultati

In questa sezione vengono presentati i metodi adottati per la validazione del processo sviluppato e i risultati ottenuti dall'analisi dei file di codice.

5.1 Scelta LLM e modalità

L'LLM adoperato per l'intero processo è meta-llama/Llama-3.2-3B-Instruct, sviluppato appunto da meta, appartenente alla collezione *Llama 3.2* e reso disponibile da Hugging Face[1]. Questa versione è progettata per essere utilizzata in compiti di tipo "instruct" (ovvero guidata da prompt specifici) e supporta l'elaborazione di un massimo di 4,096 token per ogni chiamata, suddivisi tra input e output.
Per accedere al modello, è stato utilizzato il servizio *Hugging Face Inference API*[2], un sistema che consente di inviare richieste autenticate (tramite token personale) e ottenere risposte direttamente dai modelli open-source disponibili.
Questo approccio offre diversi vantaggi:

- Eliminazione vincoli hardware: eseguire un LLM di queste dimensioni su una macchina locale richiede una potenza di calcolo media-altam con la necessità di disporre di GPU dedicata con ampie quantità di VRAM
- Semplicità d'uso e deployment immediato: non è neccassario installare localmente il modello nè configurare l'ambiente, perché l'API fornisce un'interfaccia semplice per inviare e ricevere risposte, ma soprattutto facilmente integrabile nel proprio sistema
- Ottimizzazione dei tempi di risposta: la richiesta remota comporta, ovviamente, una latenza di rete, ma il tempo complessivo rimane sempre inferiore rispetto al gestire localmente il modello (con un hardware limitato)

5.2 Valutazione

Nella Figura 5.1 viene evidenziato l'obbietivo principale, cioè valutare l'efficacia e la qualità delle model card generate automaticamente dall'LLM rispetto a quelle ufficiali presenti sulla piattaforma

[1] `https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct`
[2] `https://huggingface.co/docs/api-inference/index`

di Hugging Face. Questo confronto ha lo scopo di misurare quanto i codici creati siano pertinenti e, soprattutto, utili per gli sviluppatori finali, che si potranno affidare a questi esempi per comprendere come utilizzare correttamente e al meglio i PTM.

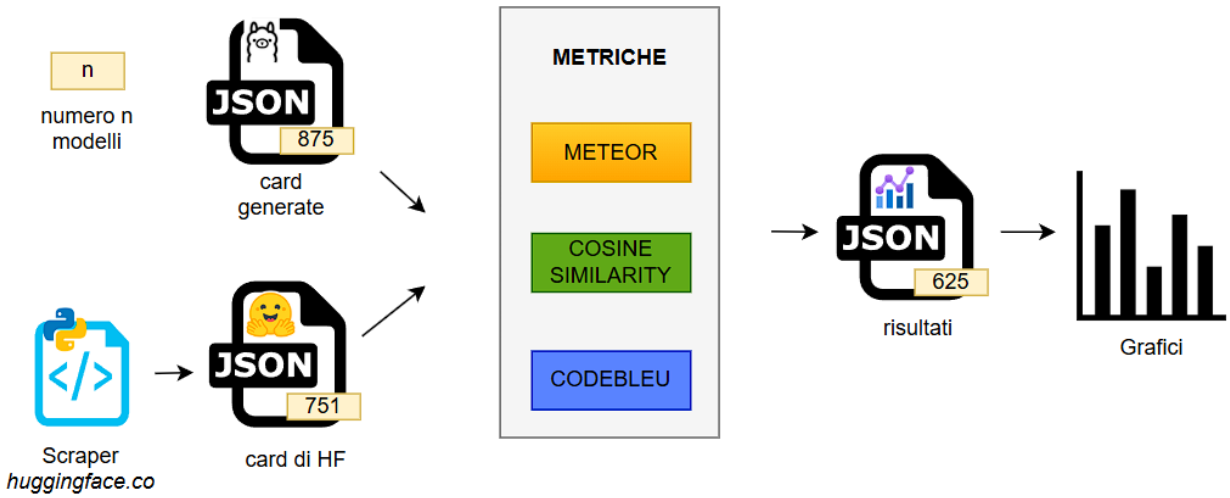

Figura 5.1: Processo di validazione

5.2.1 Costruzione dataset di test

Nonostante nel dump[1] ci fosse la feature "card data", cioè informazioni riguardanti la model card di ogni PTM, non erano presenti gli snippet di codice Python che mostrano l'utilizzo del modello specifico in un esempio di applicazione.
Quindi, per ottenere queste sezioni della card di HF, si è sviluppato un semplice script che utilizza la libreria *BeautifulSoup*[3] per effettuare web scarping delle pagine HTML dei PTM popolari selezioni per l'esperimento. Ad ogni iterazione si ricerca il tag *<code>* e quando avviene la corrispondenza si controlla che il risultato sia effettivamente appartenente al linguaggio Python, quindi semplicemente verificando che nella stringa trovata siano presenti le keyword "import", "from" e "def".
Si noti che, per ogni PTM analizzato, lo scraper potrebbe trovare anche più di uno snippet Python, perché nella stessa model card potrebbe venir illustrate diverse pipeline di utilizzo: dall'importazione con diverse librerie (per esempio *transformers* e *sentence_transformers*) fino a metodi di encoding tramite embeddings. Allo stesso tempo può accadere che non trovi nessun risultato, perché in alcuni casi specifici il codice Python potrebbe non essere inserito nella parte di model card avente il tag HTML cercato.
Un esempio di struttura JSON del dataset è rappresentato nel Listing 5.1.

[3] `https://pypi.org/project/beautifulsoup4/`

```
{
"sentence-transformers/all-MiniLM-L12-v2": [
    "from sentence_transformers...,
    "from transformers import AutoTokenizer...
    ],
"roberta-large": [
    "import pipeline...
    ],...
}
```

Listing 5.1: Esempio di struttura per le card ufficiali di HF

Dall'analisi effettuata il risultato è di 751 modelli che nelle proprie model card contengano almeno uno snippet di codice Python rispetto ai 1064 di partenza. La differenza è composta quindi da PTM che nonostante siano mediamente popolari sull'utilizzo pratico in progetti open-source abbiano una documentazione probabilmente non completa ed efficace per gli sviluppatori della community, motivando maggiormente lo scopo generale di questo studio.

5.2.2 Metriche

Dopo aver raccolto il codice Python presente nelle card ufficiali, si è proceduti nel valutare quanto fossero simili a quelle generate dall'LLM. Sono state impiegate tre metriche di confronto che valutano la somiglianza a livello lessicale, semantico e strutturale del codice:

- **METEOR** (Metric for Evaluation of Translation with Explicit ORdering)[2] è progettata principalmente per la valuzione di traduzioni automatiche, confrontando le parole secondo un matching flessibile ovvero la corrispondenza si basa su sinonimi, stemmatura e ordine con cui compaiono. Questa flessibilità permette di catturare non solo l'aspetto sintattico ma anche una parziale sensibilità semantica perché, per esempio, la definizione di una funzione di codice potrebbe essere scritta in due modi diversi ma rappresentare lo stesso significato.
  Il calcolo della metriche è il seguente:

$$\text{METEOR} = (1 - \text{penalty}) \cdot F_{\text{mean}} \tag{5.1}$$

  dove:

$$\text{penalty} = \gamma \left(\frac{\text{numero di chunk}}{\text{numero di parole corrispondenti}} \right)^{\beta} \tag{5.2}$$

$$F_{\text{mean}} = \frac{10 \cdot P \cdot R}{9P + R} \tag{5.3}$$

  - **P (Precisione)** $= \frac{\text{numero di parole corrispondenti}}{\text{numero totale di parole nel candidato}}$
  - **R (Recall)** $= \frac{\text{numero di parole corrispondenti}}{\text{numero totale di parole nella referenza}}$

  La penalità nell'equazione 5.2 considera la frammentazione e la disposizione delle parole tra il testo canditato (la card generata dall'LLM) e quello referente (card ufficiale su HF). In particolare il numero di chunk rappresenta quante sequenze di parole corrispondenti ci sono, quindi, più è frammentata l'affinità, maggiori saranno i chunk. Gamma e beta, invece, sono

parametri scelti empiricamente in base all'implementazione e bilanciano il livello della penalità che, in generale, è bassa o quasi nulla quando le parole che fanno match sono sono tutte in questa sequenza, viceversa se sono ordine sparso aumenta.
Successivamente la media armonica pesata nell'equazione 5.3 è progettata per dare più importanza alla recall rispetto alla precisione perché nelle applicazioni pratiche di traduzioni e descrizioni è spesso più dannosso non considerare informazioni importanti (bassa recall) rispetto ad includere termini e dettagli aggiungi non necessari (bassa precisione). Quindi, si preferisce considerare le descrizioni (card) potenzialmente più utili ovvero quelle che forniscono più informazioni dettagliate a costo di qualche aggiunta superflua e non strettamente necessaria, come può essere, in questo contesto, l'eventuale presenza di funzioni Python di utility che non riguardano l'utilizzo dei PTM in maniera diretta.
La metrica METEOR, in generale, non è ideale per il confronto di codice sorgente in quanto è progettata per la valuzione tra stringhe del linguaggio naturale attraverso un dizionario che non è specifico del linguaggio Python. Ma ci sono comunque dei motivi validi per considerlarla nell'esperimento:

  - parziale sensibilità semantica con l'utilizzo dei sinonimi per variabili e soprattutto per definizione di funzioni
  - l'importanza della recall per verificare che il codice generato copra tutti gli elementi del codice di riferimento (pipeline di utilizzo del PTM)
  - la penalizzazione sull'ordine delle istruzioni del codice sorgente può essere abbastanza rilevante che se in genere l'organizzazione delle istruzioni è flessbile

  Nel complesso potrebbe aiutare a catturare aspetti che altre metriche non riescono, ma deve essere considerata come parte di un multi-approccio e non come una soluzione unica ed ideale.

- **CodeBLEU** [22] è una metrica specificatamente progettata per valutare la qualità di un codice generato candidato rispetto a un riferimento e rappresenta un'estensione della metrica BLEU (Bilingual Evaluation Understudy)[17] ma adattata per catturare aspetti sintattici, semantici e strutturali di codice sorgente.
  CodeBLEU combina nel suo punteggio quattro metriche differenti:

  - *n-gram match (BLEU)* che si concentra eslusivamente sulle corrispondenze lessicali, confrontando sequenze di token (n-grammi) tra il codice generato e quello di riferimento. La formula è la seguente:

    $$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{5.4}$$

    dove:

    * p_n è la precisione degli *n-gram* di ordine n, ovvero la proporzione di n-grammi del candidato che appaiono nella referenza.
    * w_n sono i pesi assegnati a ciascun n-gram (solitamente uniformi)
    * BP (brevity penalty) è il fattore di penalità che riduce il punteggio quando la lunghezza del codice candidato è significativamente inferiore a quella del referente

Questa metrica è sensibile all'ordine dei token ma non considera la differenza semantica, quindi potrebbe avere un valore alto se i due codici con gli stessi token hanno un ordine diverso che evidenzia un comportamento diverso.

- *Weighted n-gram match* simile al funzionamento dell'n-gram match ma assegnando pesi diversi ai token in base al loro senso semantico. Ogni token viene pesato tramite un dizionario (che dipende dal linguaggio dei codici da confrontare), dando importanza a parole chiavi come, in questo contesto, "def", "return", "for", ecc. Al contrario le differenze tra i nomi delle variabili e spaziature nel codice vengono valorizzate meno rispetto alle strutture di controllo.
- *Syntax Match con AST* per valutare la similarità strutturale confrontando gli Abstract Syntax Tree dei due codici. Come menzionato nella sezione 4.3.1, l'AST permette di costruire una rappresentazione ad albero della struttura sintattica del codice dove nei nodi creati sono presenti i costrutti sintattici propri del linguaggio, mentre gli archi descrivono le relazione tra essi.
  La similarità tra i due alberi viene misurata principalmente con due metodi:
  - Tree Edit Distance (TED)[24], ovvero calcolare il numero minimo di operazioni per trasformare un albero (candidato) nell'altro
  - Difflib Sequence Matcher, ovvero confrontare la rappresentazione serializzata dell'albero tramite la libreria *difflib*[4], dove per serializzata si intende una sequenza lineare di stringhe composta da caratteri e token che descrivono un particolare costrutto. Ad esempio, per una funzione la sequenza descrive i parametri di input, il corpo e l'output.

  In entrambi i metodi il risultato viene normalizzato tra 0 e 1. Questa metrica è fondamentale perché due codici posso avere le stesse strutture logiche e quindi stesso comportamento ma sintassi differente.
- *Data flow match* confronta i flussi di dati tra variabili e funzioni cioè dove vengono definite, usate e passate tra i vari blocchi di codice. Per ogni snippet viene costruito un grafo di flusso e si confrontano le connessioni tra variabili ed operazioni riconoscendo codici che fanno la stessa cosa ma sono scritti in modo diverso.

Considerando le quattro metriche appena citate, il calcolo di CodeBLEU è descritto dalla media pesata:

$$\text{CodeBLEU} = \frac{\alpha \cdot \text{n-gram} \ + \ \beta \cdot \text{weighted n-gram} \ + \ \gamma \cdot \text{syntax} \ + \ \delta \cdot \text{data flow}}{4} \tag{5.5}$$

Dove:
α, β, γ, δ sono i pesi assegnati a ciascuna componente della metrica. In assenza di preferenze specifiche, questi pesi sono spesso uguali ovvero impostati a 1.

- Cosine similarity con Embedding: misura semanticamente la somiglianza tra due testi trasformandoli in vettori tramite modelli di embedding. Il calcolo tra due vettori **A** e **B** è definita

[4] `https://docs.python.org/3/library/difflib.html`

come:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \cdot \|\mathbf{B}\|} \tag{5.6}$$

dove:

- $\mathbf{A} \cdot \mathbf{B}$ è il prodotto scalare tra i due vettori.
- $\|\mathbf{A}\|$ e $\|\mathbf{B}\|$ sono le norme dei due vettori

Come modello di embedding viene utilizzato *CodeBERT*[9], specializzato per il codice sorgente, pre-addestrato su una grande mole di dati formati da coppie codice-testo (ad esempio, codice con relativi commenti). Il suo obiettivo è catturare la semantica del codice e rappresentarlo in uno spazio vettoriale utile per compiti come il retrieval, la generazione automatica e la valutazione della similarità tra frammenti di codice. Partendo dal processo di tokenizzazione, produce gli *hidden state* ovvero una matrice numerica risultante nel passaggio dei vari layer del *Transformer* per ogni token. Infine attraverso un processo di sintetizzazione (in questo caso utilizzando la mean pooling) si produce un unico vettore.

Il processo di calcolo delle metriche menzionate prevede di iterare sui modelli che ovviamente hanno una corrispondenza sia nel dataset delle model card di HF sia nell'output dell'LLM. I valori numerici risultanti sono salvati strutturalmente in formato JSON per una corretta leggibilità negli studi successivi.

5.3 Risultati

A partire da un insieme iniziale di **1,064 PTM**, utilizzati in **141,749 script** Python, il processo descritto nell'approccio ha portato alla generazione di **875 frammenti** di codice sintetizzati dall'LLM. Questi rappresentano i pattern ricorrenti di un sottinsieme degli snippet selezionati durante il clustering. Tuttavia, **186 PTM** non hanno prodotto alcun output valido dall'LLM, poiché gli snippet in input non rispettavano il criterio imposto dall'espressione regolare, che richiedeva la presenza esplicita del nome del modello di riferimento.
Parallelamente, il dataset contenente le sezioni di codice estratte dalle model card ufficiali di Hugging Face, tramite web scraping, ha restituito dati per **751 modelli**, un valore inferiore rispetto all'insieme iniziale di PTM analizzati, sempre pari a 1064. Pertanto, incrociando i risultati ottenuti dall'LLM con quelli delle model card e considerando solo i modelli comuni, le metriche di valutazione sono state calcolate su un totale di 625 occorrenze.

Metrica	**Valore medio**
CodeBLEU	0.30
Cosine Similarity	0.96
METEOR	0.07

Tabella 5.1: Media dei valori per ogni metrica

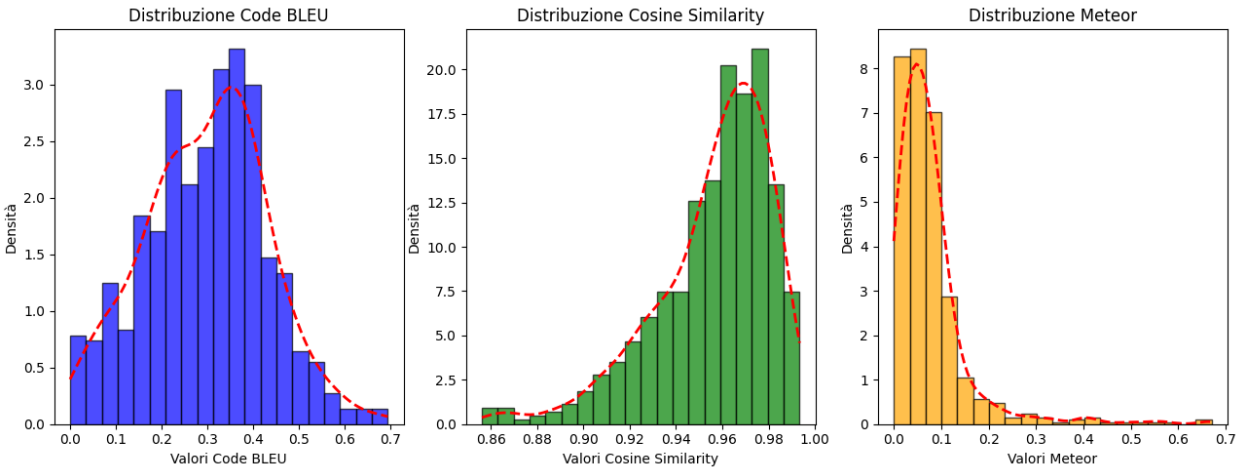

Figura 5.2: Distribuzione metriche

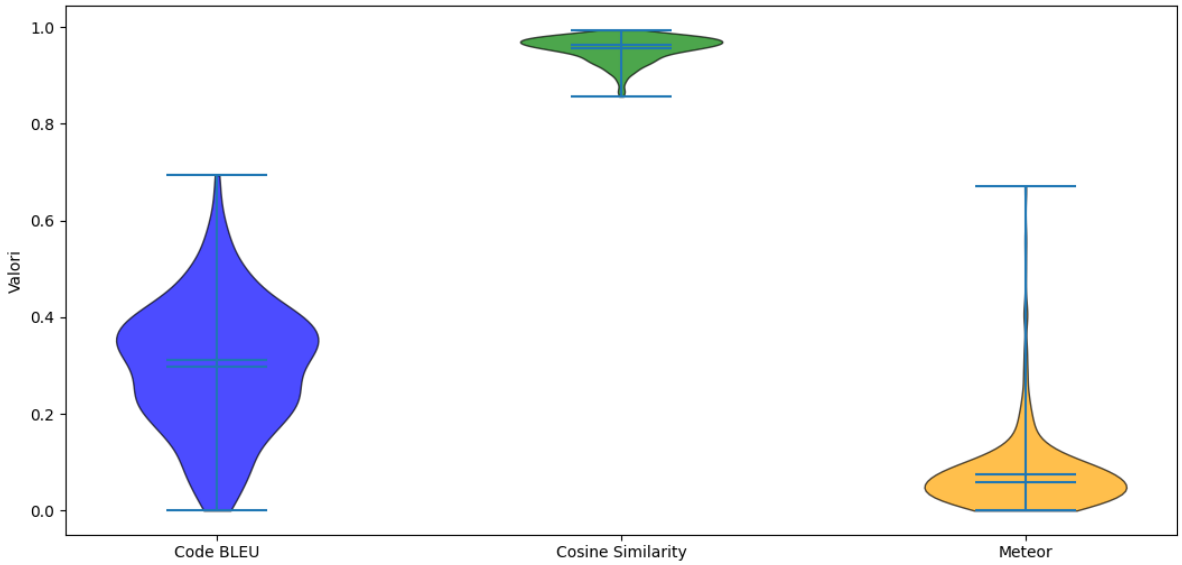

Figura 5.3: Violin plot metriche

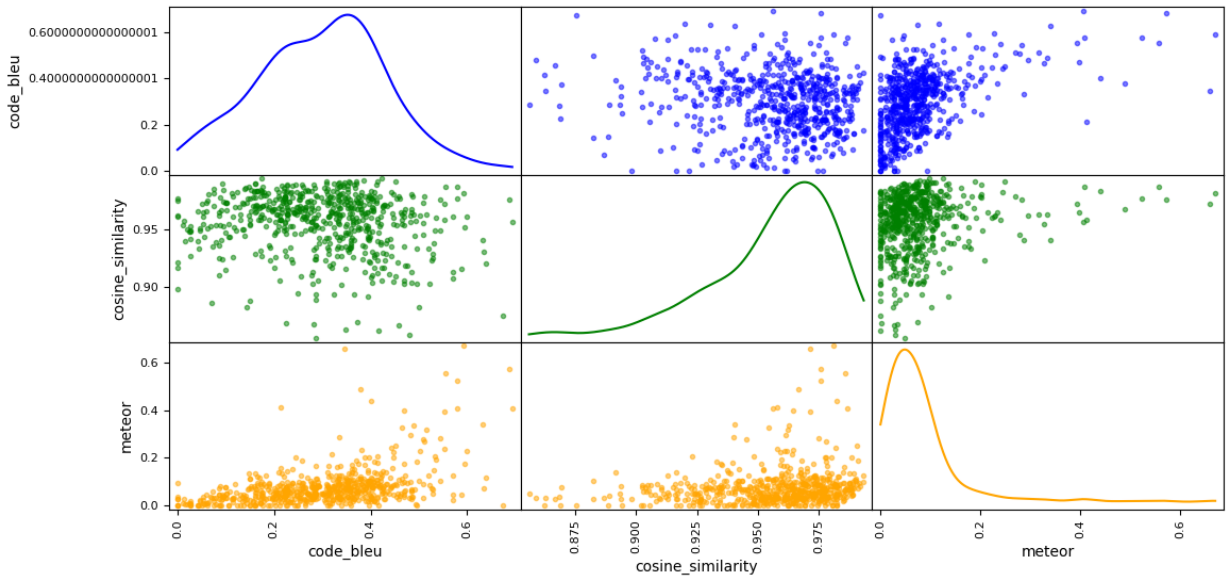

Figura 5.4: Pair plot metriche

5.3.1 Distribuzione dei valori CodeBLEU

La misurazione si basa su un'istanza in cui sono stati assegnati i seguenti pesi nell'equazione 5.5: 0.05, 0.05, 0.45, 0.45 rispettivamente (n-gram match, weighted n-gram match, syntax match, data flow match). Questa scelta è motivata dal fatto che l'obiettivo principale dell'analisi è privileggiare la corrispondenza a livello semantico e strutturale, dando molto meno valore alle scelte terminologiche dei due codici.
Osservando il primo grafico in Figura 5.2, i valori si concentrano principalmente nell'intervallo 0.2 - 0.4, con un picco attorno a 0.3. Questo suggerisce che, sebbene ci sia una certa somiglianza tra i codici generati dall'LLM e quelli delle model card, la corrispondenza non è particolarmente elevata. Il fatto che pochi valori superino 0.5, indica che le sequenze di codice sintetizzate non sono una replica diretta di quelle ufficiali, ma piuttosto una versione generalizzata dei pattern più comuni. In aggiunta, la funzione di densità stimata mostra una distribuzione non perfettamente normale, ma con una lieve asimmetria a destra. Questo conferma che la maggior parte dei valori è concentrata in una fascia medio-bassa, mentre pochi campioni raggiungono punteggi elevati.
Nel Violin Plot in Figura 5.3, la presenza di un valore mediano attorno a 0.3 conferma la tendenza già osservata con l'istogramma, evidenziando la maggior densità dei punteggi CodeBLEU nell'intervallo 0.2 - 0.4. Nel Pair Plot in Figura 5.4, si nota come i punti relativi a CodeBLEU siano distribuiti in modo abbastanza ampio, indicando una certa variabilità dei risultati. Inoltre, dal confronto con le altre metriche, non emerge una correlazione lineare particolarmente forte.

5.3.2 Distribuzione della Cosine Similarity

Qui i valori nel primo grafico in Figura 5.2 sono concentrati tra 0.8 e 0.9, con una distribuzione fortemente sbilanciata verso l'alto. Questo risultato suggerisce un'elevata sovrapposizione tra i due insiemi di dati, ma si tratta di un risultato potenzialmente *fuorviante*.
La cosine similarity tende infatti a gonfiare particolarmente i punteggi, soprattutto quando i testi confrontati condividono termini e strutture simili, anche se il significato complessivo può essere differente. Siccome la metrica mostra valori sempre molto alti, non è sempre un indicatore affidabile della qualità della corrispondenza tra i due codici.
Dal Violin Plot in Figura 5.3 si nota come l'intervallo sia compreso quasi interamente tra 0.85 e 1.0, con un picco marcato vicino a 0.9, confermando la scarsa variabilità di questa metrica. Nel Pair Plot in Figura 5.4, la distribuzione lungo l'asse della similarità coseno appare molto concentrata, e anche osservando i grafici di correlazione con CodeBLEU o METEOR, non si evidenziana una forte correlazione.

5.3.3 Distribuzione dei valori METEOR

In questo caso, invece, i valori nell'istogramma in Figura 5.2 sono molto bassi, concentrati principalmente tra 0.0 e 0.1, con pochissimi esempi che superano 0.3. Questo indica che il codice generato dall'LLM differisce in modo significativo da quello presente nelle model card ufficiali di HF, almeno secondo i criteri di METEOR. La funzione di densità stimata conferma questa tendenza, mostrando un picco netto vicino allo zero e una distribuzione fortemente sbilanciata a sinistra, con una coda molto lunga.
Tuttavia, bisogna considerare che METEOR è una metrica più adatta alla valutazione di testi in

linguaggio naturale e potrebbe non essere del tutto appropriata per confrontare frammenti di codice, dove la similarità semantica non sempre si riflette in una corrispondenza esatta tra parole e sinonimi. Anche nel Violin Plot in Figura5.3, METEOR mostra una distribuzione fortemente concentrata in prossimità dello 0, con rare eccezioni che raggiungono valori più alti. Il Pair Plot in Figura 5.4 evidenzia inoltre come la maggior parte dei punti si collochi nella parte bassa della scala METEOR, indipendentemente dai valori di CodeBLEU o della similarità coseno.

5.3.4 Considerazioni analisi comparata

Dalla Tabella 5.1 vengono mostrate le medie delle 3 metriche, inoltre osservando i tre grafici nelle Figure 5.2, 5.3, 5.3 nel loro insieme, emergono alcune considerazioni interessanti:

- Il Violin Plot in Figura 5.3 riassume in un'unica vista la distribuzione e la densità dei valori per ciascuna metrica. In particolare, si nota come CodeBLEU abbia un range più ampio, con valori che oscillano tipicamente tra 0.2 e 0.4, mentre la similarità coseno risulti costantemente elevata (attorno a 0.85-0.95) e METEOR sia invece molto basso (con la maggior parte dei valori vicini a 0.0).
- Il Pair Plot in Figura 5.4 mostra, oltre alle distribuzioni marginali sulle diagonali, i grafici di dispersione tra le coppie di metriche. Si può osservare:
  - *CodeBLEU vs. Cosine Similarity*: non vi è una correlazione lineare evidente; ciò significa che un valore alto di similarità coseno non implica necessariamente un valore alto di CodeBLEU.
  - *CodeBLEU vs. METEOR*: anche qui la dispersione è piuttosto ampia, indicando che punteggi alti o bassi di CodeBLEU non si riflettono in modo coerente in METEOR.
  - *Similarità Coseno vs. METEOR*: la maggior parte dei punti si concentra in corrispondenza di valori di coseno elevati e METEOR molto bassi, evidenziando come le due metriche misurino aspetti molto diversi del codice.

Nel complesso, i valori numerici ottenuti indicano che il codice generato dall'LLM non è una semplice copia di quello presente nelle model card ufficiali, ma piuttosto una sintesi alternativa basata su pattern più specifici, estratti direttamente dagli snippet di codice reali. Questo significa che, anche se alcune metriche automatiche risultano basse, il codice prodotto non è necessariamente inattendibile. Al contrario, questi risultati potrebbero rivelarsi particolarmente utili per gli sviluppatori umani, i quali, analizzando il codice generato, possono astrarre verso pattern più generali, cogliendo dettagli tecnici che le metriche automatiche non riescono a valutare appieno.
Inoltre in riferimento alla Motivazione, per i modelli per cui l'LLM ha prodotto un output ma non dispongono di una model card ufficiale contenente codice (**250 codici**), il materiale potrebbe essere utilizzato come base per arricchire direttamente la documentazione del modello, proponendo snippet rappresentativi del suo utilizzo tipico. Questo approccio potrebbe facilitare la creazione di model card più complete e accessibili, migliorando la fruibilità dei modelli per la community di sviluppatori.

Capitolo 6

Limitazioni

Nonostante l'approccio adottato in questo lavoro si sia dimostrato efficiente e valido nell'automatizzare l'analisi dei pattern nell'utilizzo dei Pre-trained Models esistono alcune limitazioni che hanno influenzato i risultati finali.
Un primo aspetto riguarda il *processo di selezione* dei modelli analizzati, che si basa su un criterio statistico. Inizialmente, sono stati considerati solo i modelli con un numero di file superiore a una soglia predefinita, e successivamente sono stati selezionati quelli i cui script rientravano in un determinato intervallo di lunghezza, calcolato attorno alla mediana utilizzando l'Interquartile Range (IQR). Sebbene questo approccio sia robusto, presenta alcune criticità. Da un lato, favorisce implicitamente i modelli più popolari, ossia quelli con un elevato numero di script disponibili, mentre potrebbe trascurare modelli meno diffusi ma ugualmente rilevanti per applicazioni specifiche. Dall'altro, rischia di escludere pattern d'uso rari ma significativi: alcuni modelli potrebbero presentare schemi di utilizzo molto particolari che, non rientrando negli intervalli di lunghezza selezionati, vengono così esclusi.
Un'altra limitazione riguarda la qualità e la *capacità del modello* di linguaggio utilizzato. Pur essendo un LLM di buon livello, presenta comunque dei vincoli intrinseci. La sua dimensione, con circa 3 miliardi di parametri, è inferiore rispetto a versioni più grandi della stessa famiglia (7B, 13B, 70B parametri), il che può influire sulle capacità di comprensione e generalizzazione dei pattern analizzati. Questo si traduce in risposte meno approfondite quando il codice da esaminare è particolarmente articolato. Inoltre, il modello è vincolato da una finestra di contesto limitata a 4,096 token per ogni chiamata, il che ha spesso impedito di inviare tutti gli snippet di codice desiderati. Di conseguenza, è stato necessario selezionare un sottoinsieme ridotto a poche unità, con il rischio di tralasciare informazioni rilevanti per l'analisi.
Anche le *API* impiegate hanno imposto alcune restrizioni operative. L'uso dell'*Hugging Face Inference API*, nel piano gratuito, ha comportato limitazioni sul numero di richieste effettuabili in un dato intervallo di tempo, rallentando il processo di raccolta delle risposte e obbligando a distribuirlo su più giorni. Inoltre, l'accesso a determinati modelli più potenti della piattaforma non sempre è gratuito, e un utilizzo intensivo dell'API potrebbe comportare costi aggiuntivi. Un ulteriore fattore è la variabilità nelle prestazioni, che può essere influenzata dal carico sui server di Hugging Face, con possibili rallentamenti o, in casi rari, l'indisponibilità temporanea del modello in uso.
Analogamente, l'utilizzo di *PyGithub* per l'estrazione dei dati da GitHub ha presentato ulteriori vincoli significativi. L'API di GitHub impone un rate limit di 5,000 richieste all'ora per utenti

autenticati tramite token e un massimo di 10 richieste al minuto (nel contesto di ricerca codici), rendendo necessaria una pianificazione attenta per evitare blocchi temporanei. Inoltre, ogni chiamata agli endpoint dell'API consente di recuperare un massimo di 1,000 file distribuiti tra varie repository, limitando la quantità di dati ottenibili per i modelli con molte corrispondenze nella query. Infine, alcune repository potenzialmente interessanti potrebbero essere private o avere restrizioni di accesso, rendendo impossibile l'estrazione di determinati file e riducendo la completezza del dataset raccolto.

Sempre per quanto riguarda la raccolta dati, il principali limite nella costruzione del *dataset di test*, riguarda la capacità dello scraper di individuare correttamente tutti gli snippet di codice Python presenti nelle model card.

Il fattore critico è la variazione nella struttura della pagina di quest'ultima. Sebbene nella maggior parte dei casi il codice sia racchiuso all'interno dei tag <code>, non esiste un formato standard rigido, e alcuni modelli potrebbero contenere snippet all'interno di altri elementi HTML (come <pre> o), rendendo inefficace l'estrazione automatizzata proposta. Infine, in alcune model card, gli snippet di codice potrebbero non essere direttamente visibili o potrebbero essere inclusi all'interno di immagini o documentazioni esterne collegate, rendendo impossibile la loro acquisizione tramite web scraping. Questa limitazione comporta una possibile sottostima del numero di model card con codice disponibile, influenzando la valutazione della corrispondenza tra gli snippet estratti e quelli generati dall'LLM.

Capitolo 7

Conclusione e Sviluppi futuri

Il lavoro svolto in questa tesi ha permesso di analizzare in modo sistematico la corrispondenza tra il codice riportato nelle model card ufficiali di Hugging Face e l'effettivo utilizzo dei Pre-trained Models nei progetti open-source su GitHub. Attraverso un approccio basato su data-mining, analisi automatizzata del codice e generazione di pattern mediante un Large Language Model, è stato possibile raccogliere e processare un dataset 141,749 di script Python, individuando schemi d'uso ricorrenti per un insieme di PTM pari a 875 e confrontandoli con gli esempi ufficiali forniti nella documentazione dei modelli traendo statiche per 625 di essi.
I risultati ottenuti mostrano che, sebbene esista una sovrapposizione parziale tra il codice generato dall'LLM e gli snippet presenti nelle model card, le soluzioni prodotte non costituiscono un semplice duplicato, bensì dei pattern alternativi osservati nei repository. Questo suggerisce che il codice generato possa riflettere meglio l'uso pratico dei modelli nella community, evidenziando dettagli tecnici e configurazioni non sempre documentate esplicitamente. Inoltre, l'approccio sviluppato ha evidenziato il potenziale degli LLM nell'automazione della documentazione, offrendo la possibilità di generare suggerimenti per completare le model card esistenti o per colmare le lacune nei casi in cui esse siano sprovviste di esempi pratici.
Nonostante i risultati sono stati discretamente soddisfacenti nel trovare pattern comuni, esistono diverse direzioni per migliorare e raffinare ulteriormente il lavoro svolto. Un primo aspetto riguarda il miglioramento della fase di estrazione degli snippet di codice. Attualmente, la selezione delle porzioni più rilevanti avviene tramite una combinazione di ricerca di parole chiave all'interno degli script e clustering basato su embedding. Un possibile sviluppo futuro potrebbe prevedere l'adozione di un modello di embedding più avanzato come CodeBERT, eventualmente migliorato con tecniche di fine-tuning, in modo da catturare con maggiore precisione il senso semantico del codice. Inoltre, la fase di clustering potrebbe essere resa più strutturata introducendo una determinazione dinamica del numero di cluster in base al contesto, anziché utilizzare un valore fisso, e adottando algoritmi di clustering più avanzati. Questi miglioramenti potrebbero portare a una selezione più accurata degli snippet, aumentando così la qualità e l'affidabilità dell'analisi.
Un'altra direzione di sviluppo riguarda l'estensione del processo di generazione della model card. L'esperimento condotto si è focalizzato esclusivamente sull'estrazione delle porzioni di codice presenti nelle model card dei Pre-trained Models disponibili su Hugging Face. Tuttavia, una prospettiva interessante sarebbe quella di generare un'intera model card, includendo non solo esempi di codice, ma anche sezioni descrittive fondamentali come la panoramica generale del modello, l'architettura

e i dettagli tecnici, le informazioni sui dataset di addestramento e fine-tuning, nonché i risultati dei benchmark e le relative limitazioni. Questo ampliamento richiederebbe l'integrazione nel dataset di informazione aggiuntive, come documentazioni ufficiali e articoli scientifici relativi ai modelli analizzati, oltre a una progettazione più sofisticata dei prompt, in grado di guidare l'LLM nella produzione dei contenuti richiesti.
Un ulteriore sviluppo pratico potrebbe essere l'applicazione di questo approccio nella predizione automatica delle model card. In questo scenario, il sistema potrebbe suggerire ai creatori delle repository di integrare automaticamente nella documentazione pubblica l'output generato dall'LLM nei casi in cui i modelli non dispongano di esempi di utilizzo pratico. Questo tipo di supporto sarebbe particolarmente utile per gli sviluppatori futuri, poiché consentirebbe di arricchire in modo automatizzato la documentazione con esempi di codice pertinenti e coerenti con l'effettivo utilizzo del modello. Inoltre, la standardizzazione della struttura delle model card garantirebbe che ogni modello pubblicato su Hugging Face disponga almeno di un esempio concreto di implementazione, migliorando la fruibilità della piattaforma per la community, semplificando il processo di accessibilità ai modelli.
L'analisi condotta ha quindi evidenziato l'utilità di un sistema basato su dati reali che possa contribuire al miglioramento della qualità della documentazione disponibile sulla piattaforma Hugging Face. Questo lavoro rappresenta un primo passo verso l'automazione della generazione di documentazione tecnica, fornendo una base solida per futuri sviluppi. L'obiettivo è rendere le model card più complete, strutturate e aderenti all'effettivo utilizzo dei modelli, riducendo al contempo lo sforzo richiesto agli sviluppatori nella loro redazione manuale.

Bibliografia

[1] A. Ait, J. L. C. Izquierdo, and J. Cabot. HFCommunity: A Tool to Analyze the Hugging Face Hub Community. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 728–732, Mar. 2023. ISSN: 2640-7574.

[2] S. Banerjee and A. Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[3] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Automated clustering of metamodel repositories. In *Advanced Information Systems Engineering: 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings 28*, pages 342–358. Springer, 2016.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021.

[5] D. Claudio, D. Juri, D. Davide, and P. Stefano. Codexhug: A curated dataset of hugging face pre-trained models exploited in github ecosystem. 2024.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[7] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. T. Nguyen, and R. Rubei. On the use of large language models in model-driven engineering. *Software and Systems Modeling*, pages 1–26, 2025.

[8] C. Di Sipio, J. Di Rocco, D. Di Ruscio, and S. Palombo. Cofexhug: A curated dataset of huggingface pre- trained models exploited in the github ecosystem, Dec. 2024.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[10] J. Fowkes and C. Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 254–265, 2016.

[11] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

[12] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, Dec. 2022.

[13] W. Liang, N. Rajani, X. Yang, E. Ozoani, E. Wu, Y. Chen, D. S. Smith, and J. Zou. What's documented in ai? systematic analysis of 32k ai model cards, 2024.

[14] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[15] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot, and S. Tan. Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp, 2021.

[16] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta. Recommending api function calls and code snippets to support software development. *IEEE Transactions on Software Engineering*, 48(7):2417–2438, 2021.

[17] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] M. Peeperkorn, T. Kouwenhoven, D. Brown, and A. Jordanous. Is temperature the creativity parameter of large language models?, 2024.

[20] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[21] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.

[22] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[23] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *ArXiv*, abs/1508.07909, 2015.

[24] K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, July 1979.

[25] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[27] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 319–328. IEEE, 2013.

[28] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[29] A. Williams, N. Nangia, and S. Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018.

[30] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush. Transformers: State-of-the-art natural language processing. In Q. Liu and D. Schlangen, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, Oct. 2020. Association for Computational Linguistics.

[31] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*, pages 318–343. Springer, 2009.