ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

BENCHMARKING E PROMPT TUNING DI LARGE LANGUAGE MODEL PER LA GENERAZIONE DI CODICE

Elaborato in
Programmazione Di Applicazioni Data Intensive

Relatore
Prof. Gianluca Moro

Co-relatori
Dott. Giacomo Frisoni
Dott. Luca Ragazzi

Presentata da
Luca Bighini

Quarta Sessione di Laurea
Anno Accademico 2022 – 2023

PAROLE CHIAVE

Intelligenza Artificiale

Transformer

Large Language Model

Generazione di Codice

Prompt Tuning

A chiunque mi sia stato vicino,
e mi abbia aiutato a raggiungere questo traguardo.

Abstract

Questo lavoro esplora le capacità di generazione di codice Java avanzato da parte degli attuali modelli generativi di intelligenza artificiale open source, con un focus particolare sul contesto accademico universitario. Attraverso la valutazione di tre Large Language Model (LLM) – Deepseek-Coder, Mistral e CodeLlama – questo studio introduce un nuovo benchmark derivato da compiti di programmazione avanzati richiesti a studenti laureandi. I risultati indicano che, nonostante le generali buone prestazioni, vi è un ampio margine di miglioramento, soprattutto per quanto riguarda la generazione di codice che aderisce ai principi della programmazione orientata agli oggetti e nell'uso efficace degli stream di Java. È stata inoltre studiata la tecnica del prompt tuning, rivelatasi un fattore chiave per aumentare la versatilità dei modelli. Questo lavoro suggerisce nuove prospettive per la ricerca futura, sottolineando il potenziale dell'intelligenza artificiale nel semplificare lo sviluppo software.

Introduzione

Nell'era digitale odierna, l'intelligenza artificiale (IA) sta rivoluzionando radicalmente il nostro modo di vivere e lavorare, portando a cambiamenti senza precedenti in numerosi settori. Uno sviluppo particolarmente significativo si è verificato nel campo dell'Elaborazione del Linguaggio Naturale (NLP), una sotto branca dell'AI, che ha portato alla nascita dei Large Language Model (LLM). Questi modelli hanno la capacità di eseguire vari compiti, tra i quali spicca la generazione di codice, promettendo di innovare le metodologie di sviluppo software. La capacità di generare codice a partire da descrizioni in linguaggio naturale non solo apre nuove frontiere per la programmazione, ma contribuisce anche a rendere il processo più accessibile, modernizzando potenzialmente le pratiche tradizionali di ingegneria del software.

Il presente studio si inserisce in tale contesto, con l'obbiettivo di valutare la capacità degli attuali modelli di generazione di codice open source di affrontare compiti avanzati di programmazione Java, tipici dell'ambiente accademico universitario. Questo interesse deriva dalla crescente domanda di strumenti che possano assistere gli sviluppatori software, facilitando la scrittura di codice e ottimizzando l'efficienza dello sviluppo.

Concentrandosi su questa esigenza, il presente lavoro si pone come obiettivo principale l'analisi delle prestazioni di tre LLM – Deepseek-Coder [1], Mistral [2] e CodeLlama [3] – selezionati sulla base delle loro capacità documentate e di un equilibrato numero di parametri, rappresentando così l'attuale stato dell'arte nel settore. Questi modelli sono stati valutati mediante un benchmark innovativo, appositamente creato per questo progetto di tesi, composto da esercizi avanzati di programmazione.

L'indagine si articola attorno a due aspetti fondamentali: la capacità dei modelli di implementare classi Java complesse, rispettando i principi della programmazione orientata agli oggetti, e la loro abilità nell'utilizzo di pattern consolidati e interfacce funzionali avanzate. È stata prestata particolare attenzione alla conformità del codice generato con le *"best practice"* di programmazione e all'efficacia nell'applicare tecniche avanzate, come l'uso degli stream di Java.

Attraverso un'analisi approfondita e una valutazione manuale del codice

prodotto, questa ricerca fornisce un quadro dettagliato delle capacità e dei limiti dei modelli esaminati. Emergono, in particolare, le potenzialità del prompt tuning come strumento per migliorare l'efficacia dei modelli nel risolvere compiti complessi, indicando una direzione promettente per futuri sviluppi nel campo.

Pertanto, questo studio non solo mette in luce le sfide nella comunicazione uomo-macchina per la generazione di codice di alta qualità ma apre anche a nuove prospettive di ricerca per migliorare l'interazione tra sviluppatori e modelli di IA, attraverso un utilizzo più strategico e mirato del prompt tuning.

L'esposizione del lavoro svolto è stata organizzata nel modo seguente:

- **Capitolo 1** - Vengono introdotte le nozioni teoriche fondamentali per comprendere l'intero lavoro di ricerca;
- **Capitolo 2** - Si offre una panoramica critica delle tecnologie esistenti, esaminandone limiti e problemi. Questa analisi pone le basi per identificare gli ambiti di miglioramento affrontati nel progetto di tesi;
- **Capitolo 3** - Vengono descritte le metodologie adottate per lo sviluppo del progetto, insieme alle scelte strategiche preliminari che hanno orientato la conduzione dell'esperimento;
- **Capitolo 4** - Esposizione gli aspetti tecnici legati allo sviluppo dell'esperimento, fornendo le informazioni necessarie per comprendere la sua realizzazione e le ragioni dietro le scelte operative, in modo da permetterne la replicabilità;
- **Capitolo 5** - Presentazione dettagliata dei risultati ottenuti da questo studio, seguiti da un'analisi critica che ne discute le implicazioni.

Indice

Elenco delle figure

Capitolo 1

Fondamenti Teorici

1.1 Intelligenza Artificiale

L'intelligenza artificiale (IA) è un campo dell'informatica che mira a creare sistemi capaci di eseguire compiti che richiedono intelligenza umana, come l'apprendimento, il ragionamento, la percezione, la comprensione del linguaggio naturale e l'interazione. Questa sezione esplora le basi teoriche dell'IA, mettendola a confronto con il machine learning e il deep learning, per delineare come queste tecnologie siano interconnesse e come contribuiscano allo sviluppo dell'IA moderna.

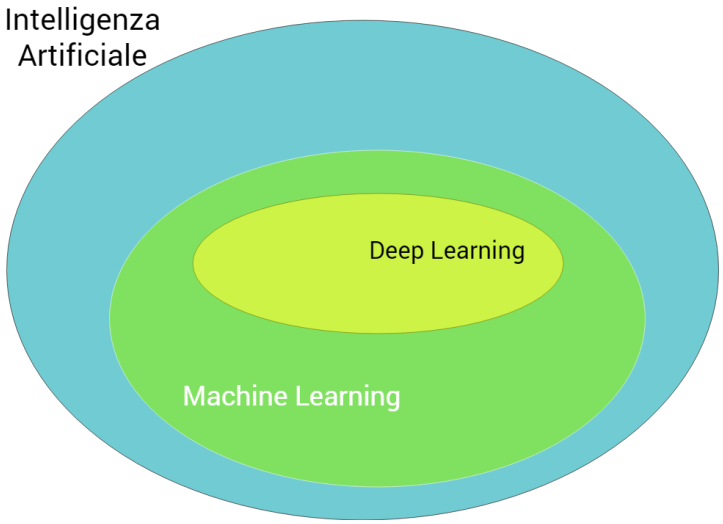

Figura 1.1: Gerarchia delle Tecnologie di Intelligenza Artificiale.

La domanda su cosa renda una macchina "intelligente" è stata al centro del dibattito sin dalla nascita dell'informatica.

Un punto di svolta nella comprensione dell'intelligenza artificiale fu l'esperimento proposto da Alan Turing nel 1950, pubblicato nella rivista Mind. Con il suo celebre test: "The Imitation Game", Turing suggeriva un criterio pragmatico per valutare l'intelligenza di una macchina: se essa può comunicare attraverso il linguaggio naturale in modo tale da non essere distinguibile da un essere umano, allora può essere considerata intelligente. Questa idea ha aperto la strada a decenni di ricerca sull'IA, focalizzandosi sullo sviluppo di sistemi capaci di emulare varie forme di intelligenza umana.

1.1.1 Machine Learning

L'intelligenza artificiale copre tutti gli approcci e le tecniche volte a simulare l'intelligenza umana nei computer. Al suo interno, il machine learning (ML) è una sottodisciplina che si focalizza sull'**abilità dei sistemi di imparare** e migliorare dalle esperienze senza essere esplicitamente programmati per ogni specifico compito. Mentre l'IA è il campo più ampio che include la simulazione di qualsiasi aspetto dell'intelligenza, il ML si concentra specificamente sull'apprendimento automatico, permettendo ai sistemi di adattarsi e migliorare le loro prestazioni man mano che vengono esposti a più dati.

Il machine learning è una tecnica nata recentemente, negli ultimi decenni del 1900, fornendo un nuovo approccio ai problemi in cui l'obiettivo è stimare, predire o ipotizzare.

Una delle definizioni più citate quando si parla di questa tecnologia, è quella proposta dal professor Tom Mitchell [11]:

> *Si dice che un programma apprende dall'esperienza E con riferimento ad alcune classi di compiti T e con misurazione della performance P, se le sue performance nel compito T, come misurato da P, migliorano con l'esperienza E.*

Cioè, riuscire a far eseguire un'attività a una macchina, che riesce in maniera autonoma a migliorare il modo in cui la esegue, usando l'esperienza che acquisisce nel farlo.

Questo approccio rappresenta una completa innovazione nel modo in cui gli algoritmi vengono utilizzati per risolvere problemi, in quanto generalmente l'approccio classico ne prevedeva lo sviluppo sfruttando la conoscenza pregressa dell'autore, mentre al contrario in questo modo la macchina impara da esempi e dati.

Alla base del machine learning c'è un **modello**. Si tratta, in sostanza, di un insieme di regole, algoritmi e parametri che lavorano insieme per interpretare i dati forniti, scoprendo legami e schemi che non sono immediatamente evidenti.

L'obiettivo è quello di rendere il modello capace di "capire" i dati in modo tale da poter fare previsioni o prendere decisioni basate su nuove informazioni.

In base alla tipologia e all'organizzazione dei dati a disposizione l'apprendimento dei modelli si suddivide in tre categorie principali:

1. **Apprendimento Supervisionato**: I dati disponibili sono già stati "etichettati", cioè per ogni esempio nei nostri dati, conosciamo l'output o il risultato che ci aspettiamo. Il modello, quindi, impara facendo correlazioni dirette tra gli input forniti e gli output noti;

2. **Apprendimento non Supervisionato**: In questo caso i dati non sono etichettati. Non c'è un risultato specifico che ci aspettiamo per ogni esempio di dati. In generale più semplici da reperire ma, il modello deve quindi trovare da solo la struttura o i pattern all'interno dei dati;

3. **Apprendimento per Rinforzo**: Il modello non sfrutta dataset pregressi, ma interagisce con un ambiente, ricevendo feedback sotto forma di ricompense per le azioni compiute, imparando così quali azioni portano ai migliori risultati.

Una fase cruciale nello sviluppo del modello è la sua validazione. Per evitare che il modello si adatti troppo specificatamente ai dati su cui viene addestrato (un fenomeno noto come overfitting), i dati disponibili vengono divisi in sottogruppi: generalmente un training set su cui il modello viene addestrato e un validation set usato per valutarne le prestazioni. Questa tecnica, conosciuta come hold-out, implica solitamente una divisione dei dati in proporzioni come 70% per l'addestramento e 30% per la validazione.

L'obiettivo di questa suddivisione è di testare la capacità del modello di generalizzare le sue predizioni a nuovi dati, non visti durante la fase di addestramento. Se il modello mostra buone prestazioni sia sul training set che sul validation set, si può considerare generalizzabile. Al contrario, se le prestazioni sul validation set sono significativamente inferiori, il modello soffre probabilmente di overfitting, indicando che è troppo specifico per i dati di training.

1.1.2 Machine Learning vs. Deep Learning

Il deep learning è un sottoinsieme del machine learning che utilizza reti neurali artificiali con molti strati per modellare complessità elevata e apprendere livelli gerarchici di informazioni dai dati. Mentre il machine learning può includere sia modelli basati su algoritmi semplici, come gli alberi decisionali e le regressioni, sia modelli più complessi, il deep learning si affida esclusivamente a

reti neurali profonde per imparare rappresentazioni sui dati a livelli di astrazione sempre più elevati. Questo consente al deep learning di eccellere in compiti come il riconoscimento visivo e la comprensione del linguaggio naturale, dove la quantità e la complessità dei dati superano le capacità dei metodi tradizionali di ML.

Il deep learning trova ispirazione nella struttura dei neuroni all'interno del cervello umano, dove esistono molti nodi, suddivisi in diversi strati (da qui il motivo della parola *deep*) che collaborano tra loro per elaborare i dati in ingresso per raggiungere risultati estremamente accurati. Nello stesso modo, all'interno delle **reti neurali** sviluppate tramite deep learning non si ritrova un solo modello da addestrare e da gestire, ma una fitta rete di *neuroni* raggruppati all'interno di strati.

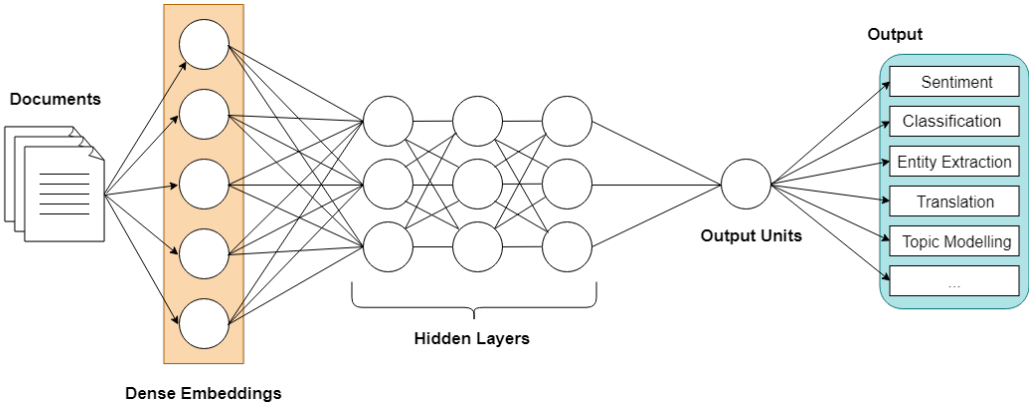

Figura 1.2: Illustrazione base di una Rete Neurale: dall'Input all'Output attraverso gli Strati Nascosti.

La divisione in strati dei neuroni consente di gestire in maniera progressiva l'astrazione dei problemi da affrontare: nei livelli più bassi si riescono a riconoscere aspetti e pattern semplici, mentre negli strati più avanzati si gestiscono gli elementi di più alto livello, così da comprendere e individuare caratteristiche sempre più complesse. Gli strati sono tra loro collegati, e le informazioni in uscita di uno sono quelle in entrata del successivo.

Ogni singolo neurone riceve in input un insieme di valori all'interno di un vettore ed elabora internamente il valore da ritornare in output. Il neurone non fa altro che eseguire la regressione per minimizzare l'errore, e raggiungere risultati sempre più precisi.

1.2 Architettura Transformer

I Transformer, introdotti da Vaswani et al. nel 2017 con il lavoro "Attention is All You Need" [4], hanno rappresentato una rivoluzione nel campo dell'elaborazione del linguaggio naturale (NLP). Prima della loro introduzione, il NLP si appoggiava prevalentemente a reti neurali ricorrenti (RNN) e Long Short-Term Memory (LSTM), che elaborano il testo sequenzialmente. Queste architetture, tuttavia, si scontravano con limitazioni significative, quali il *vanishing gradient* e la *gradient explosion*, che compromettevano l'efficacia dell'apprendimento per sequenze lunghe e complesse, rendendo difficile catturare dipendenze a lungo raggio.

A differenza delle RNN e delle LSTM, i transformer analizzano l'intera sequenza di input in modo parallelo, una caratteristica che non solo migliora notevolmente l'efficienza dell'addestramento ma anche la qualità dell'apprendimento. Il cuore di questa architettura è il meccanismo di *"Attention Multi-Testa"*, che consente ai modelli di ponderare diversamente le parti dell'input, offrendo una comprensione più fine del contesto e delle relazioni semantiche tra le parole, anche quando sono distanti tra loro.

Questa capacità di gestire dipendenze complesse e a lungo raggio ha portato allo sviluppo di modelli di intelligenza artificiale avanzati come GPT (Generative Pre-trained Transformer) [12] e BERT (Bidirectional Encoder Representations from Transformers) [13]. Questi modelli hanno stabilito nuovi benchmark in una varietà di compiti di NLP, dalla generazione di testo alla comprensione del linguaggio, dall'analisi dei sentimenti alla traduzione automatica, evidenziando l'ampia applicabilità dei transformer nel campo.

La struttura di un transformer è composta da due parti principali: l'Encoder e il Decoder.

- **Encoder**: L'Encoder è responsabile della trasformazione della sequenza di input in una serie di rappresentazioni continue che catturano le informazioni contestuali di ogni parola relativa a tutte le altre parole della sequenza. Un modello transformer è costituito da una pila di N Encoder identici.

- **Decoder**: Il Decoder prende l'output dell'Encoder e, passo dopo passo, genera una sequenza di output. Come l'Encoder, anche il Decoder è composto da una pila di N Decoder identici, che lavorano per produrre l'output finale basandosi sulle rappresentazioni fornite dall'Encoder e sui dati di output generati fino a quel momento.

Entrambi, Encoder e Decoder, sono costruiti usando una serie di componenti standard: strati di Attention Multi-Testa, strati di normalizzazione (*Layer Normalization*), connessioni residue e reti Feed-Forward. Questi componenti lavorano insieme per elaborare e trasmettere informazioni attraverso la rete.

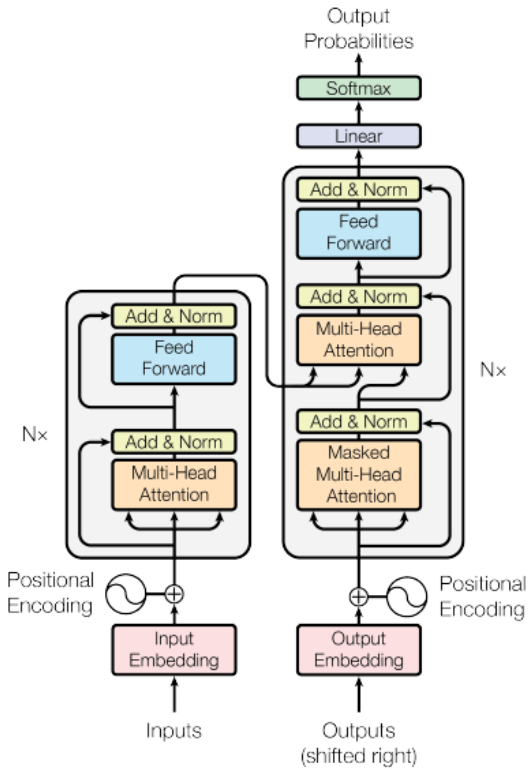

Figura 1.3: Architettura del Modello Transformer. Fonte Vaswani et al. (2017) "Attention is All You Need" [4]

Componenti di un Transformer

Come approfondito da Tong Xiao e Jingbo Zhu nel loro lavoro "Introduction to Transformers: An NLP Perspective" [14], le componenti che troviamo in un transformer sono:

- **Codifica Posizionale e Embedding di Input**
  - Embedding di Input [15]: Gli embedding trasformano i *token* di input (parole, caratteri o subword) in vettori di una dimensione fissa. Questo processo consente al modello di lavorare con dati numerici e catturare le relazioni semantiche;

– Codifica Posizionale [16]: La codifica posizionale aggiunge informazioni sulla posizione di ciascun *token* nella sequenza, consentendo al modello di considerare l'ordine dei *token*. Queste codifiche posizionali possono essere fisse o apprese durante l'addestramento e vengono sommate agli embedding di input per fornire al modello una nozione di posizione.

Questa rappresentazione viene quindi utilizzata come input per gli strati di Self-Attention.

- **Attention Multi-Testa** [17]

  L'Attention Multi-Testa è una componente cruciale che permette al modello di focalizzarsi su parti diverse dell'input per ogni *token* della sequenza.

  – Attention: Il concetto alla base dell'attention è quello di calcolare un punteggio tra ogni coppia di *token* nell'input, indicando quanto ogni *token* dovrebbe prestare attention ad ogni altro *token*, come in figura 1.4. Questi punteggi vengono poi utilizzati per pesare gli output, consentendo al modello di concentrarsi maggiormente su *token* rilevanti per ciascun *token* di input durante la trasformazione;

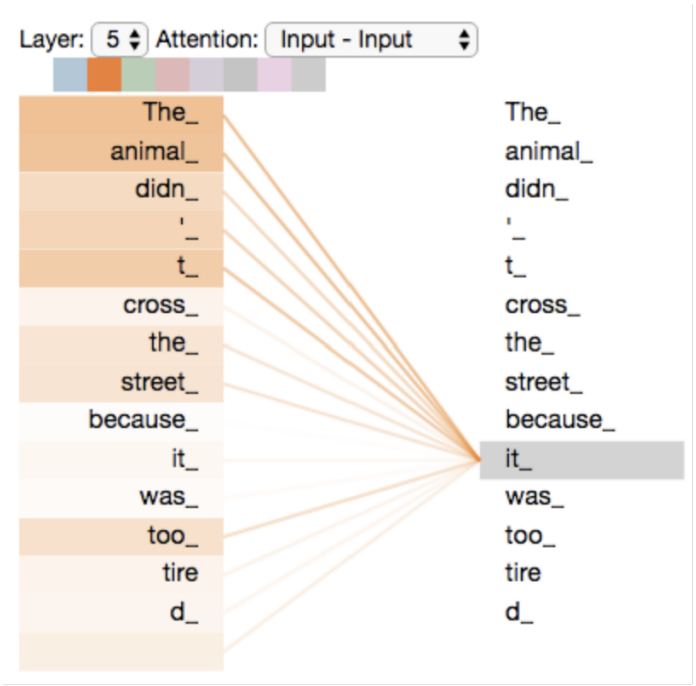

Figura 1.4: Meccanismo di Attention che Evidenzia la Relazione tra "it" e il Soggetto al quale si Riferisce. Fonte: Jay Alammar, The Illustrated Transformer

  - Multi-Testa: L'attention multi-testa divide gli embedding di input in più "teste" di attention, permettendo al modello di catturare informazioni contestuali da diverse rappresentazioni sottospaziali e punti di vista. I risultati di ogni testa vengono concatenati e trasformati per produrre l'output finale dell'attention;
  - Attention Mascherata: L'attention mascherata viene svolta solamente all'interno del Decoder e viene usata per prevenire che il Decoder "guardi avanti" ai *token* successivi nella sequenza di output, applicando una maschera ai punteggi di attention. Questo assicura che la predizione per ogni posizione possa dipendere solo dalle posizioni precedenti, mantenendo la generazione di sequenze autoregressiva.

- **Normalizzazione degli Strati e Connessioni Residue** [18]
  - Connessioni Residue: Dopo, ogni sotto-unità, come un blocco di attention o una rete feedforward, l'input di quella sotto-unità viene aggiunto al suo output. Questo aiuta a mitigare il problema del *vanishing gradient*, facilitando l'apprendimento in reti profonde;
  - Normalizzazione degli Strati: Questa operazione viene applicata dopo le connessioni residue e prima della sotto-unità successiva. Normalizza l'output di ciascun *token* attraverso la sequenza per avere media zero e varianza unitaria, contribuendo a stabilizzare la dinamica dell'apprendimento.

- **Rete Feedforward Completamente Connessa**

  Ogni strato dell'Encoder e del Decoder contiene una rete feedforward che processa ogni posizione della sequenza in modo indipendente. Questa rete è composta da due trasformazioni lineari con una funzione di attivazione non lineare interposta, tipicamente ReLU (Rectified Linear Unit) o GELU (Gaussian Error Linear Unit).

1.3 Large Language Model

I Large Language Model (LLM) sono modelli di intelligenza artificiale basati su architetture di rete neurale profonda, come i Transformer (cfr. Sezione 1.2). Addestrati su vastissime collezioni di dati, sono capaci di cogliere schemi complessi del linguaggio, consentendo loro di elaborare e generare rappresentazioni linguistiche sofisticate. Una panoramica di tecniche applicabili ai LLM viene fornita nella Figura seguente 1.5.

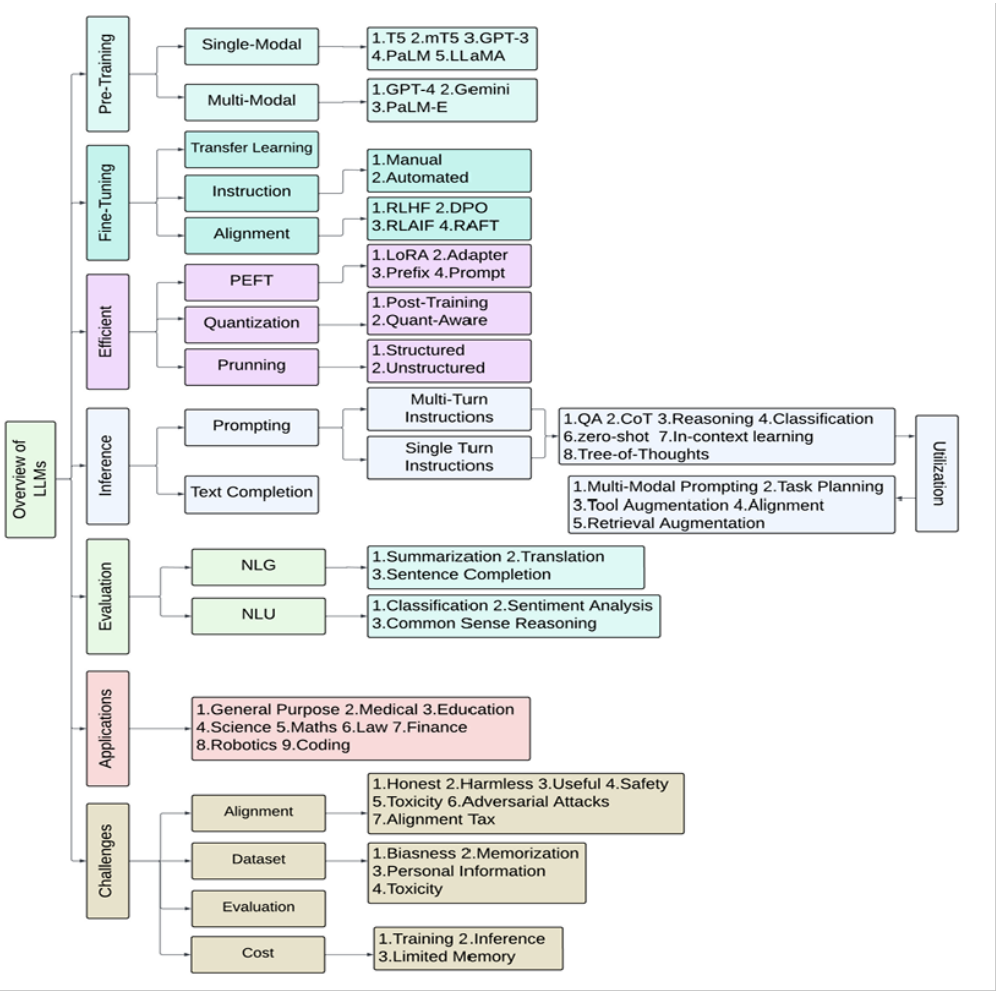

Figura 1.5: Panoramica dei LLM. Fonte: Naveed (2023) "A comprehensive overview of large language models" [5]

Fasi di Sviluppo di un LLM:

1. **Preprocessing dei Dati**

   I dati di testo vengono raccolti da diverse fonti e necessitano di essere puliti e preelaborati. Questo include la rimozione di elementi non necessari, la normalizzazione del testo (uniformare la punteggiatura, ecc);

2. **Tokenizzazione**

   La tokenizzazione, ovvero la suddivisione del testo in unità più piccole (*token*) come parole o subword è fondamentale per la creazione di un vocabolario, cosi da trasformare il testo in una forma che il modello può elaborare;

3. **Training del Modello**

   Il cuore dello sviluppo di un LLM è il suo addestramento. Ciò avviene su dataset di grandi dimensioni e richiede risorse computazionali significative. Durante il training, il modello impara a prevedere il *token* successivo nella sequenza di testo, ottimizzando i suoi parametri interni (pesi) attraverso tecniche come la backpropagation e l'aggiustamento dei gradienti. Dopo l'addestramento base, il modello può essere ulteriormente specializzato (fine-tuned) su compiti o domini specifici, migliorando le sue prestazioni su dataset più ristretti o specifici;

4. **Valutazione e Testing**

   I modelli vengono valutati su set di dati di test per misurare la loro efficacia in compiti specifici, come la comprensione del testo, la generazione di risposte, o la traduzione. Vengono comunemente utilizzate metriche come BLEU per la traduzione o F1 per compiti di classificazione;

Tra i possibili compiti di un LLM troviamo:

- Generare, completare, riassumere o correggere testo;
- Interazione e dialogo con l'utente: domanda e risposta (chatbot);
- Elaborazione di linguaggio naturale;
- Generare, completare o descrivere codice;
- Classificazione di frasi o Analisi dei Sentimenti.

Una risorsa aggiornata sui Large Language Model è fornita da Naveed et al. (2023) [5], che riassume i risultati significativi della letteratura esistente e analizza in dettaglio gli aspetti progettuali degli LLM, incluse le loro componenti architetturali, i dataset utilizzati e le strategie di addestramento impiegate.

Code LLM

Nell'ambito dello sviluppo software, i Large Language Model (LLM) per la generazione di codice stanno rivoluzionando il modo in cui gli sviluppatori interagiscono con il codice, offrendo strumenti avanzati per la codifica automatica, la comprensione del codice e l'assistenza nella risoluzione di problemi complessi. Questi modelli, come GPT (Generative Pre-trained Transformer) [12] e Codex [19], addestrati su vasti dataset di codice sorgente, hanno dimostrato capacità sorprendenti nel generare codice funzionale da descrizioni in linguaggio naturale, migliorando significativamente la produttività e l'efficienza degli sviluppatori.

Per una panoramica completa delle capacità e delle applicazioni dei LLM nella generazione di codice, si rimanda ai survey di Zhang et al. (2023) [20] e Zan et al. (2023) [21], che offrono una disamina dei progressi recenti in questo campo dinamico.

Alcuni tra i principali compiti effettuati sul codice includono: il riassunto, la generazione, la traduzione, il perfezionamento, il rilevamento di difetti e il rilevamento di cloni.

La ricerca si sta concentrando sullo sviluppo di metodologie di addestramento e valutazione più robuste, sulla creazione di dataset di valutazione dedicati e sull'esplorazione di tecniche per mitigare i bias. Inoltre, l'interazione tra sviluppatori e LLM per la generazione di codice è un'area di interesse in forte crescita.

1.4 Prompt

> *Prompt: Sequenza di testo fornita a un modello di linguaggio di grandi dimensioni per sollecitare una risposta specifica o generare continuazioni testuali coerenti, utilizzata per indirizzare l'output del modello secondo intenti definiti.*

Quindi un prompt, come definito anche da Liu et al. [22] è un insieme di istruzioni fornite a un LLM volte a migliorare e affinando le sue capacità. Un prompt può influenzare le successive interazioni con un LLM e l'output da esso generato, fornendo regole e linee guida specifiche. In particolare, un prompt stabilisce il contesto della conversazione e indica al LLM quali informazioni sono importanti e quali devono essere la forma e il contenuto dell'output desiderato.

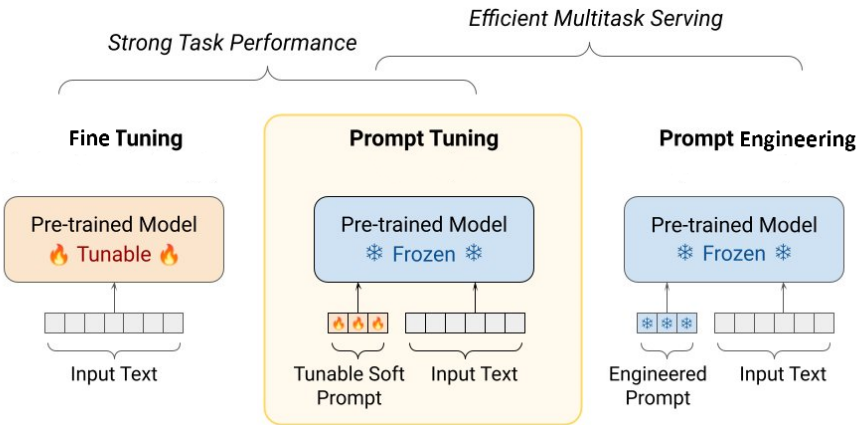

Figura 1.6: Confronto tra Differenti Approcci per Influenzare il Modello.

Nell'ambito dei LLM, esistono diversi approcci per personalizzare o guidare le risposte del modello: Fine-Tuning, il Prompt Embedding e il Soft Prompt (o Prompt Tuning), come illustrato nella Figura 1.6.

1.4.1 Fine-Tuning

Il Fine-Tuning consiste nell'adattare un modello pre-addestrato su un nuovo dataset specifico per un compito, aggiustando l'intero set di parametri del modello come mostrato in Figura 1.8. Sebbene efficace, il Fine-Tuning richiede risorse computazionali significative e può portare a problemi di dimenticanza, dove il modello perde la capacità di eseguire compiti per cui era stato originariamente addestrato.

1.4.2 Prompt Engineering

Il Prompt Engineering si riferisce alla progettazione manuale o semi-automatica di prompt che guidano il modello pre-addestrato verso un determinato compito. Nonostante la sua efficacia in certi scenari, il Prompt Engineering può essere un processo laborioso e non sempre garantisce risultati ottimali, data la sua natura spesso empirica. E' dimostrato da uno studio svolto nel 2023 che alcuni prompt pattern sono più performanti di altri nel migliorare l'output su ChatGPT [23] [24]. Di seguito, alcuni esempi di prompt pattern [25]:

- **Flipped Interaction**

  Invece di chiedere direttamente al modello di generare codice per una determinata funzionalità in Java, potresti presentare un'interazione invertita, come "Quali classi e metodi dovresti utilizzare per implementare la funzionalità di ordinamento di una lista in Java?" Questo approccio potrebbe incoraggiare il modello a considerare diverse opzioni di implementazione prima di generare il codice.

- **Recipe**

  In questa tecnica, vengono forniti passaggi dettagliati su come eseguire un'attività specifica. Ad esempio, "Creare un programma Java che legge una lista di numeri interi dall'input dell'utente e calcola la somma di tutti i numeri pari. Segui i passaggi della seguente ricetta:

  1. Crea un array per memorizzare i numeri interi.
  2. Leggi i numeri dall'input dell'utente e inseriscili nell'array.
  3. Scorri l'array e somma i numeri pari.
  4. Stampare il risultato della somma."

- **Meta Language Creation**

  La creazione di un meta-linguaggio potrebbe coinvolgere la definizione di una sintassi o di un linguaggio specifico che consente di esprimere concetti o operazioni in modo più intuitivo o astratto rispetto al codice Java "grezzo". Ad esempio, potrebbe essere creato un meta-linguaggio che semplifica la dichiarazione di classi, metodi e attributi, rendendo più agevole la generazione di codice Java corrispondente.

  "Usando il seguente meta-linguaggio semplificato, genera il codice Java per una classe che rappresenta un libro"

```
classe Libro
attributo String titolo
attributo String autore
metodo int getTitolo
```

- **Template**

  Si tratta di fornire una struttura o uno scheletro per il codice che il modello deve generare.

  "Genera una classe Java chiamata Persona con gli attributi nome e età. Includi anche un metodo per ottenere l'età della persona e un metodo per impostare il nome della persona."

```java
public class Persona {
    // Attributi
    private String nome;
    private int eta;

    // Metodo per ottenere l'eta della persona
    public int getEta() {
        // Completa il codice qui
    }

    // Metodo per impostare il nome della persona
    public void setNome(String nome) {
        // Completa il codice qui
    }

    // Altri metodi possono essere aggiunti qui
}
```

- **Fact Check List**

  Qui si fornisce un elenco di vincoli che il codice generato deve soddisfare. Ad esempio: "Genera il codice Java per un metodo chiamato trovaMassimo che accetta un array di interi e restituisce il valore massimo presente nell'array. Utilizza un elenco di controllo per assicurarti che il codice soddisfi i seguenti requisiti:"

  - Deve restituire correttamente il valore massimo presente nell'array.
  - Deve gestire correttamente gli array vuoti.
  - Deve restituire correttamente il valore massimo anche se tutti gli elementi sono negativi.

  Caso di test:

  ```
  Input: [5, 10, 3, 8, 12]
  Output atteso: 12
  Input: [-2, -5, -1, -8]
  Output atteso: -1
  Input: []
  Output atteso: Integer.MIN_VALUE
  ```

- **Reflection**

  Questa tecnica coinvolge il richiedere al modello di riflettere su un concetto o un'idea specifica nel processo di generazione del codice. "Genera un codice Java che dimostri la tua comprensione del principio di ereditarietà. Crea due classi, Veicolo e Auto, dove Auto eredita da Veicolo. Assicurati di includere almeno un attributo e un metodo in ciascuna classe e di dimostrare l'ereditarietà in azione."

- **Alternative Approaches**

  Questa tecnica coinvolge la presentazione di più modi per risolvere un problema specifico. "Sviluppa una classe Java che implementa un algoritmo di ordinamento a bolle per ordinare un array di interi in ordine crescente. Utilizza l'approccio alternativo che preferisci per implementare l'algoritmo, sia che si tratti di una variante dell'algoritmo a bolle o di un algoritmo di ordinamento diverso. Assicurati di includere commenti nel codice per spiegare il funzionamento dell'algoritmo e fornire eventuali ottimizzazioni o considerazioni particolari."

1.4.3 Prompt Tuning

Il Prompt Tuning è emerso come una tecnica efficace per personalizzare i modelli di linguaggio pre-addestrati, per compiti specifici senza necessità di modificare l'intero modello o di riformulare il dataset di addestramento. Questo approccio si distingue per la sua efficienza e per la capacità di mantenere la generalizzabilità del modello originale, come evidenziato dal lavoro di Brian Lester, Rami Al-Rfou e Noah Constant nel loro studio "The Power of Scale for Parameter-Efficient Prompt Tuning" [6].

Il Prompt Tuning si basa sull'idea di addestrare un piccolo set di parametri, specificamente quelli legati al prompt, che guidano il modello pre-addestrato verso l'esecuzione di un compito specifico, riducendo così il rischio di dimenticanza e limitando il costo computazionale. Questi prompt possono essere considerati come input iniziali o come contesti aggiuntivi che orientano le risposte del modello, influenzandone l'output senza alterarne la struttura interna o il peso dei parametri già appresi. Questo permette ad un singolo modello di essere adattato a molti più compiti diversi, come mostrato in Figura 1.7.

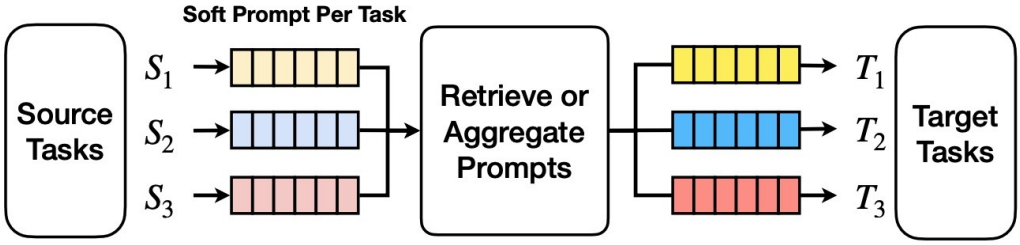

Figura 1.7: Multi-Task Prompt Tuning

La caratteristica distintiva del prompt tuning è la sua natura sia iterativa sia adattiva. Questo significa che il processo non si limita a una singola fase di addestramento, ma prevede una serie di iterazioni, ciascuna delle quali affina ulteriormente i parametri del prompt in base al feedback e alle prestazioni osservate. In questo modo, il prompt tuning adatta dinamicamente la sua strategia, ottimizzando continuamente i prompt per massimizzare l'efficacia del modello nel compito specifico.

Funzionamento del Prompt Tuning

La procedura inizia selezionando un modello pre-addestrato che sia adeguato al tipo di compito che si intende eseguire. La scelta del modello giusto è fondamentale, poiché questi sono già formati su vasti dataset e possiedono una comprensione generale del linguaggio, che serve come base da cui partire.

Successivamente, si può definire un prompt iniziale. Questo non è altro che un istruzioni formulata in modo da indirizzare il modello verso l'output desiderato. La creazione di prompt chiari e direttamente legati al compito da svolgere è cruciale per guidare efficacemente il modello.

Il cuore del processo di prompt tuning risiede nell'ottimizzazione attraverso un ciclo iterativo di un insieme ristretto di parametri, i quali vengono sintonizzati per affinare le prestazioni del modello sul compito specifico. Ciò significa che, piuttosto che rivedere l'intero modello, si concentra l'attenzione su un numero limitato di aspetti che possono essere ottimizzati per migliorare significativamente i risultati, come nella sezione di destra in Figura 1.8.

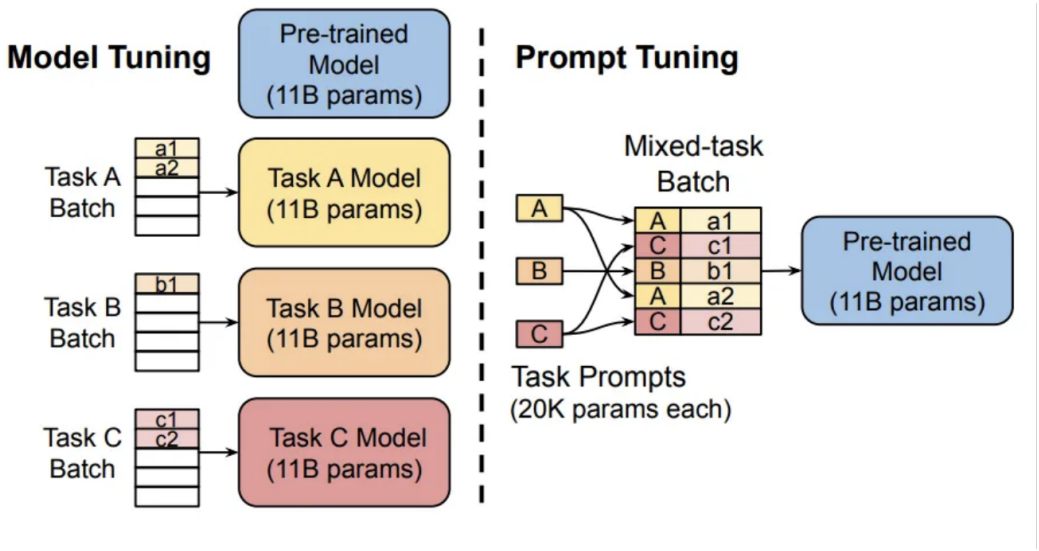

Figura 1.8: Confronto del Numero di Parametri di Addestramento fra le Tecniche di Fine-Tuning e Prompt Tuning. Fonte: Lester et. al (2021) "The power of scale for parameter-efficient prompt tuning" [6].

Una fase critica in questo processo è la valutazione e il raffinamento. Dopo ogni iterazione di addestramento, le prestazioni del modello vengono valutate per determinare l'efficacia dei prompt ottimizzati. Questo feedback è indispensabile per guidare le iterazioni future, assicurando che i prompt diventino progressivamente più efficaci nel dirigere il modello verso le risposte desiderate.

Infine, una volta che si è raggiunto un livello di prestazione soddisfacente, il modello sintonizzato tramite prompt tuning è pronto per essere implementato. Questo modello, ora ottimizzato per un compito specifico, è capace di fornire risposte precise e pertinenti, dimostrando l'efficacia di questo approccio.

1.4.4 Prompt Tuning vs. Prefix Tuning

Mentre il Prompt Tuning si concentra sull'ottimizzazione dei prompt come input iniziali per guidare il modello, il Prefix Tuning [7] consiste nell'aggiungere un prefisso "fittizio" ai dati di input durante l'addestramento. Questo prefisso, composto da *token* non visibili durante l'inferenza, viene ottimizzato per indirizzare il modello verso il compito specifico. La differenza principale risiede quindi nel meccanismo di interazione con il modello: il Prompt Tuning modifica l'input iniziale per influenzare l'output, mentre il Prefix Tuning introduce un elemento aggiuntivo (il prefisso, vedi Figura 1.9) che agisce all'interno della sequenza di input per guidare il modello.

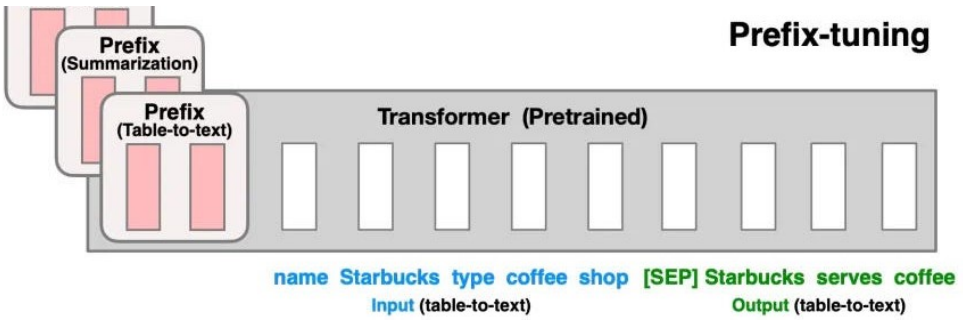

Figura 1.9: Tecnica del Prefix Tuning. Fonte: Lisa Li and Liang (2021) "Prefix-tuning: Optimizing continuous prompts for generation" [7].

1.5 Embedding Inversion

La codifica embedding trasforma le parole, frasi o documenti in vettori di numeri reali all'interno di uno spazio vettoriale continuo, superando la natura categorica e sparsa dei dati testuali. Questo processo di rappresentazione vettoriale densa è cruciale per l'efficacia dei modelli di machine learning, nell'elaborare e analizzare il linguaggio naturale.

Gli embedding sono generati attraverso l'addestramento su vasti corpus di testo, permettendo agli algoritmi di apprendere come posizionare le parole in modo che le loro distanze e direzioni nello spazio vettoriale riflettano le relazioni semantiche e sintattiche. Parole usate in contesti simili o con significati affini risultano in embedding vicini, facilitando ai modelli la cattura delle sfumature linguistiche (cfr. Figura 1.10).

L'embedding di parole può essere descritto matematicamente come una funzione di mappatura $E : W \to \mathbb{R}^d$, dove:

- W è l'insieme di tutte le parole nel vocabolario del modello,
- $\mathbb{R}^d$ rappresenta uno spazio vettoriale d-dimensionale in cui ogni parola $w \in W$ viene mappata,
- $E(w)$ è il vettore d-dimensionale corrispondente alla parola w.

La dimensione d del vettore è un parametro scelto durante la fase di progettazione del modello e rimane costante per tutti gli embeddings, indipendentemente dalla dimensione del vocabolario.

Formalmente, se consideriamo un vocabolario di dimensione V, con $V = |W|$, la funzione di embedding E può essere rappresentata come una matrice $E \in \mathbb{R}^{V \times d}$, dove ogni riga i della matrice corrisponde all'embedding d-dimensionale della i-esima parola nel vocabolario.

Per una data parola w, l'embedding è ottenuto accedendo alla riga corrispondente nella matrice di embedding E. Se w_i denota l'indice della parola w nel vocabolario (cioè, w è la i-esima parola in W), allora l'embedding di w è il vettore $E(w_i) = E_i$, dove E_i indica la i-esima riga della matrice E.

Questo processo trasforma le parole da rappresentazioni sparse, tipicamente codificate come *one-hot vectors* in uno spazio di alta dimensionalità V, a rappresentazioni dense in uno spazio molto più ridotto $\mathbb{R}^d$, facilitando il trattamento computazionale e l'apprendimento di relazioni semantiche e sintattiche profonde tra le parole.

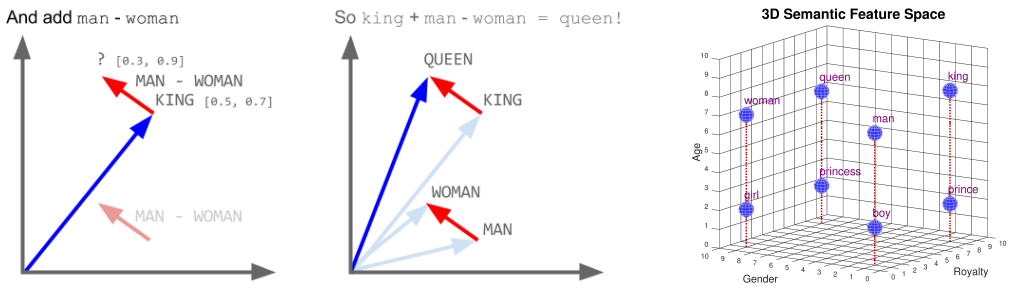

Figura 1.10: Collocazione delle Parole nello Spazio - Word Embedding.

Queste rappresentazioni sono ottenute tramite algoritmi come Word2Vec, GloVe, o strati di embedding in reti neurali, con il vantaggio di ridurre la dimensionalità dei dati pur conservando informazioni contestuali ricche. All'interno di un LLM, gli embedding svolgono un ruolo fondamentale già nella fase

iniziale di trattamento dei dati: le parole inserite nel modello vengono convertite in vettori di embedding, che poi vengono elaborati attraverso vari strati del modello, come i meccanismi di attention nei Transformer, per produrre output, effettuare previsioni, o eseguire specifici compiti di NLP.

Il concetto di **inverse embedding** [26] si riferisce al processo di mappatura inversa da queste rappresentazioni vettoriali (embedding) a parole o frasi comprensibili, un processo che potrebbe teoricamente permettere di decodificare i prompt ottimizzati a una forma leggibile e interpretabile dall'uomo. Questo concetto è particolarmente rilevante per comprendere fino a che punto le informazioni contenute negli embedding possano essere ricche e fedelmente rappresentative del testo originale.

Definizione:
Data una funzione di embedding $F : T \to \mathbb{R}^d$ che mappa un token di testo x in uno spazio vettoriale d-dimensionale, l'operazione di inverse embedding cerca di definire una funzione inversa $F^{-1} : \mathbb{R}^d \to T$ tale che $F^{-1}(F(x)) \approx x$.

$$y = F(x)$$

dove y rappresenta l'embedding vettoriale del token x, e l'obiettivo è trovare $F^{-1}(y)$ che si avvicini il più possibile a x.

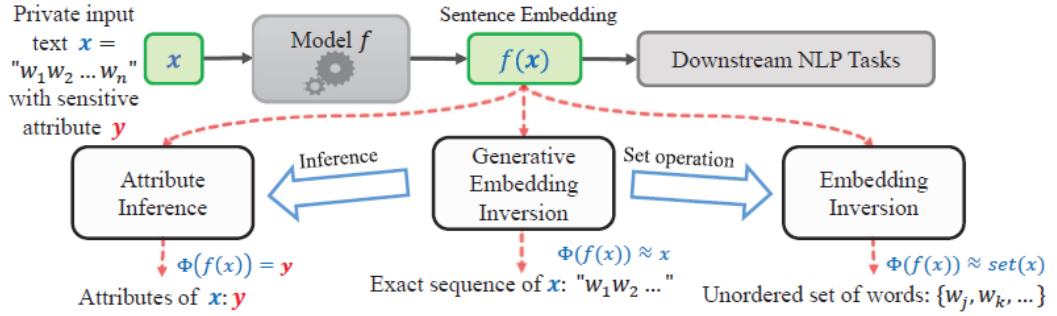

Figura 1.11: Panoramica degli attacchi di inversione di embedding e inferenza di attributi sui modelli linguistici. Entrambi gli attacchi possono essere condotti sull'embedding della frase f(x) [8].

L'inverse embedding si riferisce a un processo iterativo il cui obiettivo è trovare una funzione (cfr. Figura 1.11) che mappi in modo ottimale un dato embedding indietro nel suo spazio originale o in uno spazio target, al fine di ricostruire l'informazione o l'oggetto originale, come illustrato nella Figura seguente 1.12.

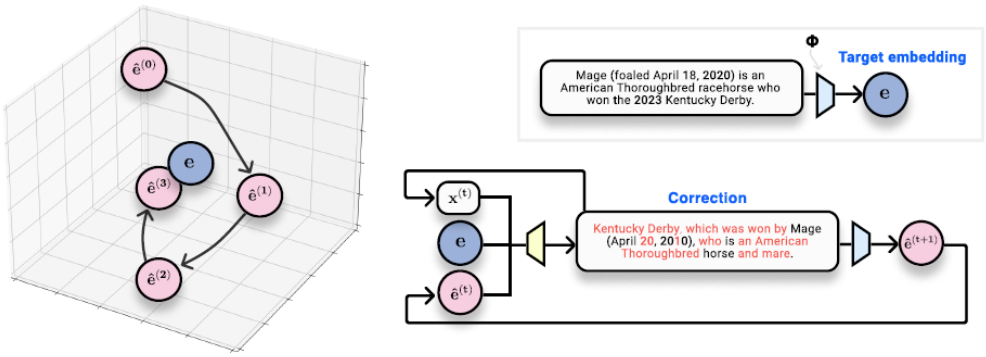

Figura 1.12: Processo Iterativo per la Ricostruzione dell'Embedding. Fonte: Morris et. al (2023) "Text Embeddings Reveal (Almost) As Much As Text" [9].

Il processo iterativo inizia con la scelta di un embedding. Si stabilisce quindi l'obiettivo della ricostruzione, definendo il formato o lo spazio target in cui si intende ricostruire l'entità originale. Successivamente, si sceglie una funzione o modello in grado di "tradurre" l'embedding di ritorno nello spazio originale o in uno spazio target prefissato. Questa funzione di mappatura può basarsi su diverse tecniche, come modelli di deep learning, algoritmi di machine learning o specifici metodi matematici.

Dopo aver identificato la funzione di mappatura, il processo procede con una fase di ottimizzazione iterativa, durante la quale la funzione viene perfezionata per migliorare la qualità della ricostruzione.

Segue una fase di valutazione, in cui si misura la qualità della ricostruzione confrontando gli elementi ricostruiti con i loro equivalenti originali o target.

Il ciclo di ottimizzazione e valutazione si ripete fino al raggiungimento di un livello di precisione nella ricostruzione ritenuto soddisfacente. Questo metodo iterativo assicura un affinamento continuo della funzione di mappatura, migliorandola progressivamente per ottenere la ricostruzione più fedele possibile dell'entità originale.

L'inverse embedding è un processo complesso che richiede una comprensione profonda dello spazio di embedding e dello spazio target, nonché delle tecniche matematiche e computazionali per mappare efficacemente tra questi spazi. La sua efficacia dipende dalla qualità dell'embedding originale e dalla capacità della funzione di mappatura di catturare e invertire le relazioni nascoste nei dati codificati.

Sfide nell'Inverse Embedding

1. **Perdita di Informazioni:** Gli embeddings possono comportare una perdita di informazioni, rendendo l'operazione di inversione non esatta o ambigua.

2. **Ambiguità Semantica:** La polisemia e la condivisione di embedding simili produce *token* con significati diversi che complicano la ricostruzione precisa dell'input originale.

3. **Dimensionalità e Complessità:** La mappatura inversa da uno spazio ad alta dimensionalità a un vocabolario discreto è complessa.

Questo processo, sebbene teoricamente possibile, è complesso nella pratica. Tra le tecniche più moderne troviamo: **PatchScopes** [27], un framework che sfrutta le avanzate capacità generative dei LLM per tradurre le informazioni codificate nelle rappresentazioni nascoste in linguaggio naturale. PatchScopes riesce ad avere un approccio più flessibile e dettagliata negli strati iniziali del modello, introducendo la possibilità di utilizzare un modello più espressivo per analizzare le rappresentazioni di un modello più piccolo. Non richiede un addestramento specifico e può beneficiare di un vocabolario aperto, rendendolo estremamente versatile. Metodi di interpretazione precedenti basati sulla proiezione delle rappresentazioni nello spazio del vocabolario e sull'intervento nella computazione del LLM possono essere considerati casi specifici di questo framework.

L'idea chiave di Patchscopes è sfruttare le avanzate capacità dei LLM nella generazione di testo simile a quello umano per tradurre le informazioni codificate nelle loro rappresentazioni nascoste.

Se fosse possibile realizzare un meccanismo efficace di inverse embedding, ciò aprirebbe notevoli prospettive per la comprensione e la generazione del linguaggio. Per esempio, potrebbe migliorare significativamente la capacità dei modelli di IA di decodificare informazioni complesse da rappresentazioni vettoriali, facilitando la generazione di testo più coerente, preciso e contestualmente appropriato. Questo avrebbe applicazioni in diversi ambiti, come la traduzione automatica, dove la capacità di ricostruire accuratamente il significato da embeddings potrebbe portare a traduzioni di qualità superiore.

Inoltre, il successo nell'inverse embedding potrebbe migliorare la capacità dei modelli di effettuare inferenze a partire da testi codificati, potenziando sistemi di risposta a domande, sintesi di testi e assistenti virtuali. Ciò potrebbe anche facilitare la spiegabilità dei modelli di IA, permettendo una maggiore comprensione di come le decisioni vengano prese basandosi sulle rappresentazioni interne.

Ad esempio, nel prompt tuning, il modello apprende una serie di embedding ottimizzati che funzionano efficacemente come input per guidarlo nella generazione di risposte o nell'esecuzione di compiti specifici. Questi embedding, tuttavia, non corrispondono direttamente a parole o frasi nel linguaggio umano, sono piuttosto un insieme di valori numerici che il modello "comprende". Se potessimo applicare con successo l'inverse embedding a questi prompt codificati, potremmo tradurre questi valori numerici di nuovo in testo naturale, fornendo una finestra sul "pensiero" del modello: quali caratteristiche del prompt lo guidano verso determinate risposte o comportamenti.

Capitolo 2

Revisione della Letteratura

2.1 Code Large Language Model

L'evoluzione dei modelli di linguaggio basati su codice (Code LLM) ha segnato un cambiamento significativo nel modo in cui la programmazione e l'automazione del codice vengono percepite e utilizzate. A partire dal rilascio di Codex da parte di OpenAI nel 2021 [19], abbiamo assistito a un rapido sviluppo in questo campo, con una serie di innovazioni che hanno migliorato la capacità dei modelli di comprendere e generare codice. Questo percorso ha portato alla nascita di diversi modelli chiave, ciascuno contribuendo con unici punti di forza e specializzazioni.

Codex ha rappresentato una pietra miliare essendo uno dei primi modelli ad altissima capacità di generazione di codice. La sua abilità di tradurre istruzioni in linguaggio naturale in codice funzionante in una vasta gamma di linguaggi di programmazione ha aperto nuove possibilità, come la creazione di GitHub Copilot, uno strumento che offre suggerimenti di codice in tempo reale agli sviluppatori.

La distinzione tra i modelli open source (mostrati in Tabella 2.1) e quelli closed source è diventata particolarmente interessante nell'ultimo anno. I modelli open source hanno adottato innovative tecniche di ottimizzazione e apprendimento, consentendo loro di offrire prestazioni notevolmente migliorate. Questi avanzamenti hanno permesso ai modelli open source di avvicinarsi, e in alcuni casi, eguagliare i punteggi di modelli ben più grandi e più noti come GPT, che opera con un numero di parametri significativamente maggiore.

Con circa 30 miliardi di parametri riusciamo ad ottengono punteggi simili ai modelli proposti da OpenAi:

- GPT-3 [28]: Lanciato nel 2020 con circa 175 miliardi di parametri
- GPT-4 [29]: Rilasciato a luglio 2023 con circa 1 trilione di parametri

Model	Win Rate	humaneval-python	java	javascript	cpp
CodeFuse-DeepSeek-33b	43.58	76.83	60.76	66.46	65.22
DeepSeek-Coder-33b-instruct	42.42	80.02	52.03	65.13	62.36
DeepSeek-Coder-7b-instruct	41.5	80.22	53.34	65.8	59.66
Phind-CodeLlama-34B-v2	39.96	71.95	54.06	65.34	59.59
CodeLlama-70b-Instruct	36.04	75.6	47.2	57.76	48.45
DeepSeek-Coder-33b-base	35.58	52.45	43.77	51.28	51.22
WizardCoder-Python-34B-V1.0	35.42	70.73	44.94	55.28	47.2
CodeLlama-70b	35.38	52.44	44.72	56.52	49.69
DeepSeek-Coder-7b-base	32.17	45.83	37.72	45.9	45.53
CodeLlama-34b-Instruct	31.5	50.79	41.53	45.85	41.53
WizardCoder-Python-13B-V1.0	31.35	62.19	41.77	48.45	42.86
CodeLlama-34b	30.88	45.11	40.19	41.66	41.42
WizardCoder-15B-V1.0	29.46	58.12	35.77	41.91	38.95
CodeLlama-13b-Instruct	28.42	50.6	33.99	40.92	36.36
CodeLlama-13b	26.73	35.07	32.23	38.26	35.81
CodeLlama-7b-Instruct	24.23	45.65	28.77	33.11	29.03
CodeShell-7B	22.85	34.32	30.43	33.17	28.21
CodeLlama-7B	22.85	29.98	29.2	31.8	27.23
OctoCoder-15B	21.69	45.3	26.03	32.8	29.32
Falcon-180B	21.2	35.37	28.48	31.68	28.57
CodeLlama-7b-Python	21.15	40.48	29.15	36.34	30.34
StarCoder-15B	21.12	33.57	30.22	30.79	31.55
StarCoderBase-15B	20.69	30.35	28.53	31.7	30.56
CodeGeex2-6B	17.96	33.49	23.46	29.9	28.45
StarCoderBase-7B	17.38	28.37	24.44	27.35	23.3
WizardCoder-3B-V1.0	16.27	32.92	24.34	26.16	24.94
CodeGen25-7B-multi	15.88	28.7	26.01	26.27	25.75
DeepSeek-Coder-1b-base	14.67	32.13	27.16	28.46	27.96
StarCoderBase-3B	12.19	21.5	19.25	21.32	19.43
WizardCoder-1B-V1.0	10.88	23.17	19.68	19.13	15.94
CodeGen25-7B-mono	8.69	33.08	19.75	23.22	18.62
StarCoderBase-1.1B	8.65	15.17	14.2	13.38	11.68
CodeGen-16B-Multi	7.62	19.26	22.2	19.15	21
StableCode-3B	6.58	20.2	19.54	18.98	20.77
Phi-1	6.5	51.22	10.76	19.25	14.29
DeciCoder-1B	6.35	19.32	15.3	17.85	6.87
SantaCoder-1.1B	5	18.12	15	15.47	6.2

Tabella 2.1: Confronto tra Code LLM su Diversi Benchmark - HuggingFace Big Code Models Leaderboard

2.1.1 Problematiche

Gli LLM, specialmente quando applicati alla generazione di codice, affrontano diverse sfide e problematiche. Ecco le principali:

- Difficoltà nella comprensione del contesto specifico e delle sottigliezze dei requisiti di programmazione senza un'adeguata personalizzazione;
- L'aderenza alla sintassi specifica del linguaggio di programmazione target, soprattutto per i modelli addestrati su più di un linguaggio;
- La sicurezza del codice generato che può introdurre vulnerabilità o bug di sicurezza non intenzionali;
- Difficoltà a generalizzare correttamente a partire da pochi esempi o specifiche vaghe. I modelli richiedendo spesso input dettagliati per generare codice funzionale che vada a coprire anche i casi limite;
- I compiti di programmazione particolarmente complessi o che richiedono una profonda comprensione del dominio possono risultare difficili per i LLM, soprattutto se il codice da generare richiede l'uso di librerie o dipendenze esterne.

Queste sfide sono attivamente esplorate dalla comunità di ricerca e da aziende che lavorano sullo sviluppo di LLM.

2.1.2 Nuove Tecniche introdotte da AlphaCodium

Nel gennaio 2024, Tal Ridnik, Dedy Kredo e Itamar Friedman [10] hanno introdotto AlphaCodium, un'evoluzione del precedente modello AlphaCode [30], che rappresenta un notevole progresso nel campo della generazione automatica di codice. AlphaCodium si distingue per il suo approccio innovativo, che incorpora un flusso di lavoro iterativo basato su test e orientato al codice, finalizzato al miglioramento delle prestazioni dei LLM nella risoluzione di problemi di codifica. Questo modello affronta alcune delle sfide più significative associate alla generazione di codice. Il processo di AlphaCodium è articolato in diverse fasi:

1. **Pre-elaborazione**: dove il modello riflette sul problema, descrivere il problema in forma di elenco puntato, affrontando l'obiettivo del problema, gli input, gli output, le regole e i vincoli e altri dettagli rilevanti presenti nella descrizione del problema. Ragiona sui test pubblici, spiegando il motivo per cui ciascun input del test porta all'output;

2. **Fasi Iterative**: il modello genera e valuta le soluzioni di codice, utilizzando sia test pubblici che test generati artificialmente per affinare e perfezionare le soluzioni proposte, secondo questo procedimento:

    - Generare possibili soluzioni, le classifica e sceglie la migliore, in termini di correttezza, semplicità e robustezza;
    - Generare test IA aggiuntivi per cercare di coprire casi e aspetti non coperti dai test pubblici originali (cfr. Figura 2.1).

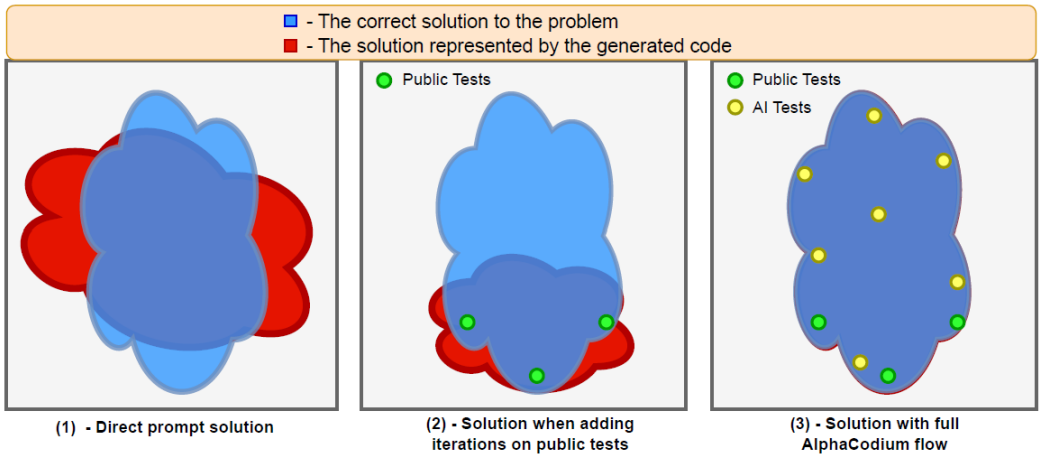

Figura 2.1: Aggiunta di Test Generati dall'IA al Test Set. [10]

    - Genera una prima soluzione al problema iniziale. Sarà proprio da questa che inizierà il ciclo iterativo di esecuzione-correzione. Ogni soluzione viene eseguita sui test, fino a quando i test passano o quando viene raggiunto un limite di tentativi. Ogni volta che il codice fallisce su un test specifico, si cerca di correggerlo, dato il messaggio di errore e generare la successiva soluzione.

Una strategia fondamentale di AlphaCodium è la progressione graduale dalla comprensione e analisi del problema alla generazione e iterazione di soluzioni di codice, evitando decisioni premature e incoraggiando un'esplorazione approfondita di diverse soluzioni potenziali. Questo approccio è ulteriormente rafforzato dall'uso di "Test Anchors", che aiutano a distinguere tra codice errato e test inaccurati, garantendo che le soluzioni finali siano robuste e affidabili.

2.2 Soft Prompting nella Generazione di Codice

Il soft prompting, a differenza dei tradizionali prompt testuali che si affidano all'inserimento manuale di testo per guidare i modelli nella generazione di output specifici, introduce l'uso di *token* virtuali, i cui valori sono ottimizzati per specifici task. Questo approccio si distingue per la sua capacità di adattare dinamicamente il comportamento del modello alle necessità specifiche di generazione di codice, senza richiedere un'esplicita riformulazione del prompt da parte dell'utente.

Le tecniche attuali di soft prompting possono essere suddivise in diverse categorie, in base al loro approccio:

- **Prompt Tuning**: descritto nella Sezione 1.4.3
- **Inserimento di Embedding**: Una variante del soft prompting prevede l'inserimento diretto di embedding vettoriali nello spazio di input del modello. Questi embedding, ottimizzati durante un processo di addestramento, funzionano come prompt dinamici che guidano la generazione del modello. In applicazioni di generazione di codice, ciò permette di modulare finemente il contesto di generazione in base a variabili quali lo stile di codifica e le specifiche di implementazione.
- **Continuous Prompting**: Il continuous prompting estende ulteriormente il concetto di prompt tuning, permettendo l'ottimizzazione continua dei prompt in base al feedback ricevuto durante la generazione di codice. Questa tecnica si rivela particolarmente efficace in scenari di sviluppo software iterativo, dove i requisiti possono evolvere rapidamente.

La tecnica di soft prompting ha trovato applicazione in una varietà di contesti, tuttavia, nell'ambito della generazione di codice, si è tradizionalmente optato per l'utilizzo di prompt statici, creati manualmente. Questa differenza di approccio pone un interrogativo rilevante sull'applicabilità e l'efficacia del prompt tuning nel dominio specifico della generazione di codice. La flessibilità offerta dalla capacità di adattare i prompt in modo dinamico potrebbe infatti facilitare l'integrazione di questi modelli in processi di sviluppo software agili, caratterizzati dalla necessità di adattarsi rapidamente a nuove esigenze o modifiche.

Capitolo 3

Modellazione del Progetto

In questo capitolo, si mira a presentare un'analisi approfondita e sistematica del metodo proposto, il quale è finalizzato a incorporare in modo efficiente sia il Benchmarking sia il Prompt Tuning nei LLM per ottimizzare la generazione di codice. La necessità di sviluppare metodologie innovative in questo ambito emerge dall'esigenza di superare le limitazioni attuali nella generazione automatica di codice, descritte nella sezione 2.1.1.

3.1 Creazione del Dataset

La fase di creazione del dataset ha rivestito un'importanza fondamentale all'interno del progetto di tesi, il cui obiettivo era indagare le capacità dei LLM nell'ambito della generazione di codice per la programmazione orientata agli oggetti (OOP) in Java. Per valutare e testare efficacemente le prestazioni dei modelli, è stato necessario sviluppare un dataset appositamente pensato per questo scopo, il quale è stato ospitato su Hugging Face Hub[1].

Hugging Face emerge come una piattaforma di spicco nel settore dell'intelligenza artificiale. Dal 2016, ha giocato un ruolo chiave nel rendere accessibili le tecnologie IA, mettendo a disposizione una ricca libreria di modelli preaddestrati oltre a molte libreria software. La piattaforma incentiva, inoltre, un'innovazione aperta attraverso "Hugging Face Hub", che si configura come uno spazio collaborativo dove ricercatori e sviluppatori possono condividere e cooperare su progetti.

Il dataset creato per il nostro studio è attualmente privato per garantire la riservatezza e l'integrità dei materiali didattici utilizzati. Questo dataset comprende una selezione di 42 esercizi scelti tra le prove d'esame del corso di OOP offerto dall'Università di Bologna, coprendo un arco temporale che

[1] `https://huggingface.co/`

va dal 2019 al 2023. Questa collezione mira a riflettere le sfide tipicamente affrontate da studenti e professionisti. La decisione di focalizzarsi su questi specifici compiti d'esame è stata guidata dalla volontà di offrire un insieme di dati significativo e rappresentativo delle varie tipologie di problemi, assicurando al contempo una gestione efficiente del dataset.

Il dataset è organizzato per ciascun compito d'esame in cinque colonne, garantendo l'assenza di valori nulli o mancanti. Le specifiche di queste colonne sono dettagliate di seguito:

1. **Num_es**: un identificatore univoco per ogni esercizio, composto dall'anno e dal numero dell'esercizio, che facilita il riferimento e l'organizzazione dei dati;

2. **Name**: il nome dell'interfaccia da implementare, fornendo al modello un'indicazione precisa sulla classe da creare, cruciale per l'esecuzione dei test correlati;

3. **Interfaces_methods**: dettagli sulle interfacce e i metodi richiesti per la soluzione dell'esercizio, specificando le tecniche e le funzionalità da implementare. Questa sezione include di base tre interfacce, che servono come input per il modello. La prima è progettata per indicare l'oggetto che i metodi devono restituire, assicurando che ci sia coerenza nel tipo di dato fornito in uscita. Segue un'interfaccia che deve essere implementata dalla classe risultante dall'output, dove ciascun metodo viene descritto brevemente per chiarirne la funzione. Infine, si introduce un'interfaccia di utilità chiamata Pair, progettata per facilitare la gestione di oggetti formati da coppie di dati. Per un esempio di codice si faccia riferimento al Listato 3.1;

4. **Class**: la soluzione di riferimento proposta dall'insegnante;

5. **Test**: una serie di test unitari progettati per verificare la correttezza delle soluzioni generate, indispensabili per assicurare che le implementazioni rispettino i criteri di adeguatezza. Per un esempio di codice si faccia riferimento a Listato 3.2.

```
public interface Tree<E> {
    E getRoot();
    List<Tree<E>> getChildren();
    Set<E> getLeafs();
    Set<E> getAll();
}

public interface TreeFactory {

    /**
     * Creates a tree consisting of a single node (root) without any children.
     * This method is intended to generate a basic tree structure with only one element, serving as the
         root.
     *
     * @param <E> the type of the elements stored in the tree
     * @param root the value of the root node
     * @return a Tree instance representing a single node with the specified root value
     */
    <E> Tree<E> singleValue(E root);

    /**
     * Generates a tree with a specified root and exactly two children.
     * This method allows for the creation of a simple tree structure where the root node directly
         connects to two child nodes.
     *
     * @param <E> the type of the elements stored in the tree
     * @param root the value of the root node
     * @param child1 the first child tree of the root
     * @param child2 the second child tree of the root
     * @return a Tree instance with the specified root and two child trees
     */
    <E> Tree<E> twoChildren(E root, Tree<E> child1, Tree<E> child2);

    /**
     * Creates a tree with a single root and multiple single-valued children.
     * This method is designed to construct a tree where the root node has several children, each of
         which is a leaf node.
     *
     * @param <E> the type of the elements stored in the tree
     * @param root the value of the root node
     * @param children a list of values, each corresponding to a single child of the root
     * @return a Tree instance representing a root with several single-valued child nodes
     */
    <E> Tree<E> oneLevel(E root, List<E> children);

}

Only if necessary use:
public interface Pair<X,Y> {
  X getX();
  Y getY();
  int hashCode();
  boolean equals(Object obj);
}
```

Listato 3.1: Esempio di Interfaccia.

```
@org.junit.Test
public void optionalTestToStrings() {
  Tree<String> tree = this.factory.singleValue("a");
  assertEquals("a[]", tree.toString());

  Tree<Integer> tree2 = this.factory.oneLevel(10, List.of(11,12,13));
  assertEquals("10[11[], 12[], 13[]]", tree2.toString());

  Tree<Integer> treeA = this.factory.oneLevel(10, List.of(11,12));
  Tree<Integer> treeB = this.factory.singleValue(20);
  Tree<Integer> treeC = this.factory.twoChildren(1, treeA, treeB);
  assertEquals("1[10[11[], 12[]], 20[]]", treeC.toString());

  Tree<Integer> tree3 = this.factory.chain(10, List.of(20,30));
  assertEquals("10[20[30[]]]", tree3.toString());

}

@org.junit.Test
public void testSingleValue() {
  // un albero con solo la radice
  Tree<String> tree = this.factory.singleValue("a");

  assertEquals("a", tree.getRoot());
  assertTrue(tree.getChildren().isEmpty());
  assertEquals(Set.of("a"), tree.getLeafs());
  assertEquals(Set.of("a"), tree.getAll());
}
```

Listato 3.2: Esempio di Test

Di seguito viene mostrata una rappresentazione visiva di come è organizzato il dataset

	num_es	name	interfaces_methods	class	test
0	2019 - a01a	GraphFactory	public interface Graph<X> {\n Set<X> getNod...	\npublic class GraphFactoryImpl implements Gra...	\n@org.junit.Test\npublic void testDirectedCha...
1	2019 - a02a	WorkflowsFactory	public interface Workflow<T> {\n Set<T> get...	\npublic class WorkflowsFactoryImpl implements...	\n@org.junit.Test\npublic void testSingleTask(...
2	2019 - a02b	ExamsFactory	public interface CourseExam<A> {\n Set<A> a...	\npublic class ExamsFactoryImpl implements Exa...	\n@org.junit.Test\npublic void testSimpleExam(...
3	2019 - a03a	RWControllersFactory	public interface RWController {\n int enter...	\npublic class RWControllersFactoryImpl implem...	\n@org.junit.Test\npublic void testPermissiveR...
4	2019 - a03b	PostOfficeFactory	public interface PostOffice {\n enum Operat...	\npublic class PostOfficeFactoryImpl implement...	\nprivate void basicChecks(PostOffice po) {\n ...

Figura 3.1: Visualizzazione della Struttura del Dataset

Il dataset ha dunque svolto un ruolo chiave nel testare la capacità del LLM di produrre codice che non solo aderisse alle specifiche tecniche fornite ma fosse anche funzionale e corretto, secondo i criteri stabiliti dai test unitari predefiniti.

3.2 Test di LLM nella Generazione di Codice

3.2.1 Metodologia

Nel progetto di valutazione delle capacità di generazione di codice da parte dei LLM, è stata adottata una metodologia mirata a testare la loro efficacia nella creazione di classi Java che implementano metodi definiti nelle interfacce fornite. Il processo si è basato sull'utilizzo di un prompt fisso, al quale venivano concatenate le interfacce Java in input. Questa procedura ha permesso ai modelli di generare, in output, classi completamente funzionali che implementano i metodi specificati dalle interfacce, seguendo le descrizioni e i requisiti dati.

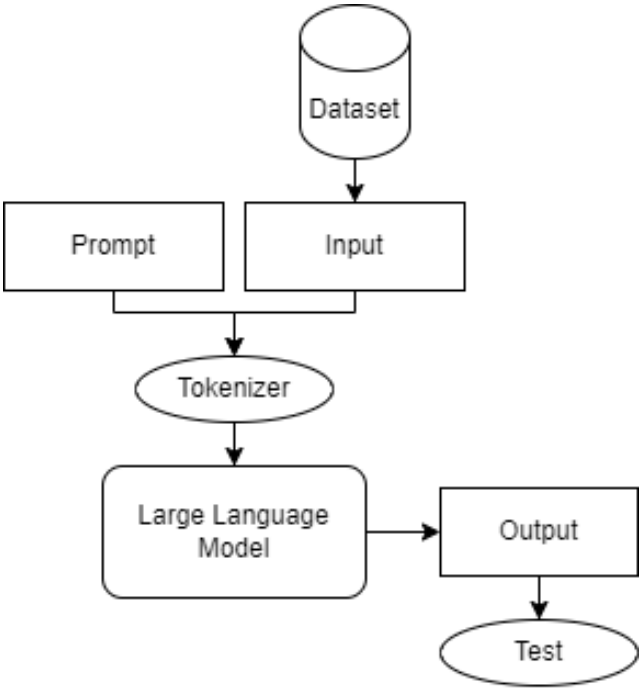

Figura 3.2: Diagramma del flusso di lavoro

Per assicurare la validità e la rilevanza dei test è stato impiegato un benchmark personalizzato, descritto approfonditamente nel paragrafo precedente 3.1. Questo benchmark, composto da una serie di compiti derivati da esami passati di programmazione OO, funge da standard rigoroso per valutare le prestazioni, l'efficacia e la qualità dei modelli. L'uso di criteri ben definiti e rilevanti è fondamentale per confrontare accuratamente diversi LLM tra loro e per monitorare i progressi nel tempo.

3.2.2 Analisi dei Modelli

I Large Language Model presi in esame in questo lavoro sono:

- **DeepSeek-Coder-6.7b-Instruct** [1]: modello aggiornato a gennaio 2024, si posiziona come un punto di riferimento nell'evoluzione delle tecniche di apprendimento automatico avanzato per affrontare le sfide inerenti alla generazione e alla comprensione del codice. Questo avanzamento rappresenta un salto qualitativo nell'ambito degli strumenti di intelligenza artificiale, riducendo le distanze tra i modelli di codice open-source e quelli proprietari. Caratterizzato da un addestramento su un corpus di 2 trilioni di *token* provenienti da 87 linguaggi di programmazione, DeepSeek-Coder beneficia di una vastissima base di dati che facilita una comprensione approfondita e versatile delle diverse sintassi e paradigmi di programmazione.

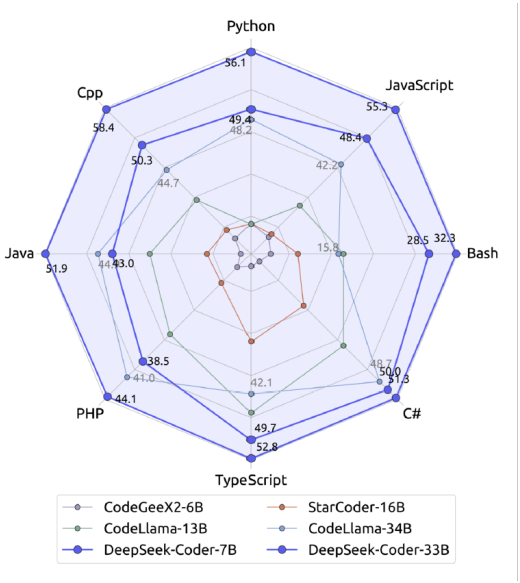

Figura 3.3: Prestazioni dei Modelli su diversi Linguaggi di Codice [1]

Il modello implementa strategie di addestramento, come la predizione del *token* successivo e l'innovativo approccio Fill-in-the-Middle (FIM), per affinare le sue capacità di integrare codice in maniera logica e funzionale nei frammenti preesistenti. Il FIM, attraverso le sue varianti PSM (Prefix-Suffix-Middle) e SPM (Suffix-Prefix-Middle), introduce una tecnica di pre-addestramento che sfida il modello a riorganizzare e completare frammenti di codice in modi nuovi e complessi, migliorando significativamente la sua flessibilità e efficacia.

L'architettura del modello, basata sul Transformer è arricchita da ottimizzazioni specifiche come l'Embedding di Posizione Rotatorio (RoPE) e l'Attention Query-Raggruppata (GQA), progettati per massimizzare l'efficienza e la precisione del modello.

- **Mistral-7B-Instruct-v0.2** [2]: pubblicato ad ottobre 2023, è un modello di linguaggio naturale avanzato, che dimostra come sia possibile ottenere elevate prestazioni mantenendo al contempo un'efficienza di inferenza, superando modelli precedenti più grandi come Llama 2 13B e Llama 34B, specialmente in ambiti come la matematica e la generazione di codice. Questo modello utilizza tecniche innovative di attention, quali l'Attention Query-Raggruppata (GQA) e l'Attention a Finestra Scorrevole (SWA), per velocizzare l'inferenza, ridurre i requisiti di memoria, e gestire sequenze più lunghe a costi computazionali minori. Mistral 7B, che opera su un'architettura Transformer, è progettato per facilitare l'implementazione sia localmente che su piattaforme cloud, e per essere facilmente personalizzabile per una vasta gamma di compiti. Mistral 7B mira a bilanciare alte prestazioni con efficienza per modelli di linguaggio di grandi dimensioni, rendendoli più accessibili e pratici per un'ampia varietà di applicazioni reali.
- **CodeLlama-7b-Instruct-hf**: Code Llama [3], rilasciata a gennaio 2024, si specializzata nella generazione e nel completamento del codice, che si è evoluta dal progetto Llama 2. Integra un addestramento su larga scala e fine-tuning mirato, offre prestazioni superiori in compiti quali la sintesi di codice, il completamento, il debugging e la creazione di documentazione. Caratterizzato dall'uso di modelli preaddestrati, supporta contesti fino a 100.000 *token*, fine-tuning per infilling e istruzioni specifiche. Sono disponibili varianti che coprono esigenze generali e specifiche di Python, ogni modello è ottimizzato per massimizzare sicurezza e aderenza alle istruzioni umane. L'addestramento utilizza l'ottimizzatore AdamW, con una programmazione coseno del tasso di apprendimento e un approccio al fine-tuning che preserva il tasso di apprendimento originale per risultati ottimali.

Sono stati scelti i seguenti modelli in quanto, a parità di numero di parametri, hanno ottenuto prestazioni superiori su diversi benchmark e tali che che potessero essere utilizzati sull'hardware a disposizione.

3.2.3 Metodo di Valutazione

La valutazione dei modelli è stata strutturata attorno a due aspetti: una quantitativa e l'altra qualitativa.

Prendendo ispirazione dalla metrica Pass@k [19], comunemente utilizzata per valutare la capacità di modelli di intelligenza artificiale di generare soluzioni corrette in k tentativi, è stato applicato un approccio simile per calcolare il tasso di successo al primo tentativo dei modelli nel superare i test JUnit inclusi nel dataset illustrato nella Sezione 3.1. Questo approccio **quantitativo** ci ha fornito una valutazione oggettiva delle prestazioni di ciascun modello, offrendo indicazioni precise sulla loro efficacia nell'eseguire le funzionalità definite con successo. Inoltre, è stata svolta una valutazione **qualitativa** del codice generato, focalizzandosi su aspetti come la pulizia, l'efficienza, la manutenibilità e l'aderenza alle *best practices* di OOP in Java, fornite dal corso di studi, per determinare l'efficacia con cui queste sono state implementate. La natura soggettiva e complessa di tali criteri ha reso necessaria una valutazione manuale, al fine di cogliere appieno le sfumature della qualità del codice prodotto.

3.3 Prompt Tuning applicato a LLM

Nel contesto delle strategie di prompting precedentemente esaminate nella Sezione 1.4, il Prompt Tuning emerge come una metodologia particolarmente efficace per affinare i modelli di apprendimento automatico. Questa tecnica è stata applicata a DeepSeek-Coder [3.2.2], un modello all'avanguardia per la generazione di codice. L'obiettivo era verificare se l'introduzione di soft prompt potesse effettivamente migliorare le prestazioni del modello in specifici compiti di programmazione.

Un prompt testuale iniziale è stato selezionato per fornire al modello un contesto del compito. Invece di modificare l'architettura del modello, si è optato per l'inserimento di input mirati, con l'intento di raffinare il prompt fornito in maniera iterativa, così da allinearsi meglio al contesto e alle esigenze del compito. Attraverso questa pratica, il prompt ha mostrato di poter migliorare e adattarsi in modo dinamico agli input ricevuti, aumentando così l'efficienza di DeepSeek-Coder nella creazione di output pertinenti.

Per procedere con l'addestramento, il dataset è stato diviso in due parti: due terzi sono stati dedicati all'allenamento e il terzo rimanente per i test. La divisione è stata fatta estraendo un elemento ogni tre per il test set, anzichè in modo casuale, poiché esercizi vicini tendono a essere simili. Dopo l'addestramento, il modello è stato testato con gli input non visti del test set per misurare la precisione e l'adeguatezza del prompt ottimizzato. La valutazione finale ha incluso un'analisi comparativa tra le versioni di DeepSeek-Coder, con e senza l'applicazione del Prompt Tuning, per determinare l'impatto concreto di questa tecnica sulle prestazioni del modello.

Capitolo 4

Sviluppo

4.1 Ambiente di Sviluppo

Il progetto è stato sviluppato utilizzando Google Colab, una piattaforma di sviluppo basata su cloud che consente di scrivere, eseguire e condividere codice Python in modo interattivo. Google Colab si basa sulla tecnologia dei Jupyter Notebooks, offrendo un'interfaccia utente intuitiva che permette agli sviluppatori di creare e gestire documenti contenenti codice e testo.

Uno degli aspetti più significativi di Google Colab è la sua capacità di eseguire singoli blocchi di codice, o "celle", indipendentemente, facilitando il testing incrementale e il debugging, senza dover eseguire l'intero script ogni volta.

Google Colab fornisce accesso a risorse hardware di alto livello, tra cui CPU, memoria RAM, storage su disco e GPU, in particolare, l'ambiente utilizzato include:

CPU

Una CPU Intel(R) Xeon(R) CPU @ 2.00GHz, che appartiene alla famiglia dei processori Intel di sesta generazione con modello 85. Questo processore integra una memoria cache di 39 MB, opera ad una frequenza di 2 GHz, garantendo prestazioni elevate per applicazioni che richiedono molta potenza di calcolo.

Il processore supporta l'esecuzione parallela su 4 core fisici per un totale di 8 thread grazie alla tecnologia Hyper-Threading (HT), particolarmente utile per applicazioni di calcolo intensivo, come simulazioni scientifiche, elaborazione di grandi set di dati e applicazioni di machine learning.

Memorie

Una memoria RAM di 50 GB.
Una memoria di archiviazione su disco di 201 GB.

GPU

Una GPU Tesla V100-SXM2-16GB progettata da NVIDIA, incorpora l'architettura Volta GV100. Dotata di 16 GB di memoria HBM2 e di 5120 CUDA Cores, consente di eseguire una vasta gamma di operazioni di calcolo in parallelo. La sua compute capability di 7.0 garantisce la compatibilità con le ultime versioni del CUDA Toolkit, permettendo agli sviluppatori di sfruttare pienamente le capacità di questa GPU per ottimizzare le prestazioni.

4.2 Setup Sperimentale

Installazione e caricamento delle librerie necessarie:

```
!pip install datasets huggingface_hub
!pip install torch transformers accelerate peft
!pip install wandb
```

- `datasets (v2.17.1)`: di Hugging Face fornisce un modo facile e efficiente per caricare e pre-elaborare dati, dando accesso a centinaia di dataset pubblici in vari formati, ottimizzati per la ricerca e l'uso nell'apprendimento automatico.

- `huggingface_hub (v0.20.3)`: permette agli utenti di condividere e accedere a modelli pre-addestrati e dataset su Hugging Face.

- `torch (v2.1.0)`: è una libreria di machine learning open source. La sua principale caratteristica è il supporto per i tensori, che sono strutture di dati ottimizzate per il calcolo ad alte prestazioni, e per le reti neurali dinamiche, che permettono una maggiore flessibilità nella progettazione dei modelli.

- `transformers (v4.37.2)`: di Hugging Face offre una vasta collezione di modelli pre-addestrati, facilmente scaricabili e utilizzabili.

- `accelerate (v0.27.2)`: è una libreria di Hugging Face che permette di eseguire facilmente script di machine learning e deep learning su hardware accelerato (come GPU e TPU) con minimi cambiamenti al codice.

- `peft (v0.8.2)`: è una libreria per adattare in modo efficiente modelli pre-addestrati di grandi dimensioni senza mettere a punto tutti i parametri di un intero modello. Questa libreria verrà utilizzata per svolgere Prompt Tuning.

- `wandb (v0.16.3)`: è una piattaforma software che fornisce strumenti per il tracciamento degli esperimenti di machine learning, la visualizzazione dei dati e la gestione del workflow di progetto. Supporta la registrazione di metriche di addestramento in tempo reale, la configurazione degli esperimenti, i log dei risultati, e la comparazione tra diversi run di addestramento. wandb è progettato per aiutare a monitorare, confrontare e ottimizzare i modelli di machine learning in modo efficiente.

Caricamento del modello e del Tokenizer

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

model_name = #Sostituisci con il nome del modello

tokenizer = AutoTokenizer.from_pretrained(model_name,
                                          trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"

model = AutoModelForCausalLM.from_pretrained(model_name,
          use_cache = False,
          trust_remote_code=True,
          torch_dtype=torch.bfloat16).cuda()
```

Al parametro `"model_name"` vengono assegnati i modelli descritti nella Sezione 3.2.2: `"deepseek-ai/deepseek-coder-6.7b-instruct"`,
`"mistralai/Mistral-7B-Instruct-v0.2"`,
`"codellama/CodeLlama-7b-Instruct-hf"`.

Il parametro `"trust_remote_code"` indica che si è disposti ad accettare e eseguire codice remoto che potrebbe essere caricato insieme al tokenizer o al modello.

I parametri `"tokenizer.pad_token = tokenizer.eos_token"` e `"tokenizer.padding_side = "right"` impostano il token di padding a *End Of Sequence* (EOS) e specificano che deve essere applicato sulla destra delle sequenze di token.

Il modello è spostato sulla GPU per accellerare i calcoli e sfruttare il parallelismo cuda. Inoltre, con il paramentro `"torch_dtype = torch.bfloat16"` si imposta il tipo di dato a bfloat16, che è un formato a precisione ridotta che consente di risparmiare memoria.

Definizione del prompt e dell'input

> *"You are an accomplished Java developer tasked with creating complex, generic data structures and ensuring they meet rigorous interface specifications.*
> ***@@ Instruction***
> *Your job is to create a Java class named* `{name}Impl` *that implements the* `{name}` *interface. Use the following interface:*
> `{interface}`
> *In response, implement all methods of the* `{name}` *interface. When possible, use Java patterns and compact code with streams.*
> ***@@ Response****"*

Il prompt in questione è stato concepito con l'intento di fornire direttive precise al modello pre-addestrato, al fine di modulare il suo comportamento in modo specifico. Tale prompt è articolato in tre sezioni:

La prima sezione si prefigge l'obiettivo di delineare la figura con la quale il modello è chiamato a emularsi.

La seconda (*Instruction*) è dedicata alla descrizione delle istruzioni da svolgere. In particolare, questa parte del prompt specifica l'input variabile, rappresentato dalle informazioni assegnate dinamicamente alle variabili "name" e "interface".

La terza sezione (*Response*) prevede la risposta del modello, ossia l'output generato a seguito dell'elaborazione delle istruzioni ricevute.

```
from datasets import load_dataset
from huggingface_hub import notebook_login

notebook_login()

dataset = load_dataset("BigoLuca/dt_java", split=['train'])[0]
prompt_template = prompt.format(name=name,
                                interface=interfaces_methods)
```

Per preparare l'input da fornire al modello, è necessario autenticarsi sulla piattaforma Hugging Face e procedere con il caricamento del dataset (una rappresentazione di quest'ultimo è illustrata nella Figura 3.1).

Successivamente, per ogni singolo esame, le variabili presenti nel prompt vengono sostituite con i dati specifici tratti dal dataset, utilizzando il comando "format". Una volta ottenuto il prompt aggiornato, esso viene tokenizzato con il tokenizer configurato precedentemente, rendendolo pronto per essere elaborato dal modello.

Il testo di input per il modello si distingue per la sua notevole estensione, causata non solo dal prompt iniziale, ma soprattutto dall'inclusione delle interfacce, che mediamente presentano tre o quattro metodi ciascuna. Ogni metodo è accompagnato da una documentazione dettagliata che ne descrive le funzionalità e enumera i parametri necessari, allo scopo di semplificare l'apprendimento del modello per un determinato compito. In media, la lunghezza degli input sono 799 parole o token. Questa dimensione testimonia la diversità del corpus di addestramento, con una lunghezza massima registrata di 1119 e una minima di 395 token. Un esempio di input è fornito al Listato 3.1. Invece, gli output target, definiti dal professore presentano una media di 722 token, ma variano significativamente in lunghezza, con estremi che vanno da 242 a 1235 token. È importante considerare che l'uso di modelli diversi potrebbe incontrare limitazioni relative alla gestione di input di questa lunghezza. I modelli utilizzati per questo studio soddisfano ampiamente le dimensioni richieste, poiché presentano una *contex length* di addestramento rispettivamente di 16K per DeepSeek-Coder, 8K per Mistral e 100K per CodeLlama.

4.3 Inferenza e Testing

```
output = model.generate(input_ids=input, max_new_tokens=1500)
print(tokenizer.decode(output[0], skip_special_tokens=True))
```

Questo frammento di codice illustra il processo di inferenza utilizzando un modello pre-addestrato. Il metodo `generate` del modello prende come parametri:

- `input_ids`: l'input tokenizzato, creato in precedenza, che rappresenta la sequenza di *token* da cui generare la risposta.
- `max_new_tokens`: il numero massimo di nuovi *token* da generare, in questo caso impostato a 1500, limitando così la lunghezza della risposta generata.

La risposta generata dal modello, rappresentata da una serie di *token*, viene quindi decodificata in linguaggio naturale tramite il metodo `decode` del tokenizer, escludendo i *token* speciali attraverso l'opzione `skip_special_tokens=True`.

Successivamente, il codice generato, che assume la forma di una classe Java (un esempio è riportato al Listato 5.3), viene trasferito manualmente in Visual Studio Code, dove avviene la compilazione e l'esecuzione della classe `Test.java`, che utilizza la libreria JUnit per eseguire test automatici. Attraverso JUnit, è possibile verificare il numero di test superati dalla classe generata. Questo passaggio richiede un intervento manuale poiché la classe generata potrebbe non essere sempre direttamente compilabile.

4.4 Addestramento del Prompt

Per affinare le prestazioni del modello selezionato, "DeepSeek-Coder", mediante il prompt tuning, abbiamo utilizzato la libreria `peft`, che fornisce strumenti specifici per la personalizzazione dei modelli di intelligenza artificiale basati su Transformer.

```
from peft import get_peft_model, PromptTuningInit,
                        PromptTuningConfig, TaskType

peft_config = PromptTuningConfig(
    task_type=TaskType.CAUSAL_LM,
    prompt_tuning_init=PromptTuningInit.TEXT,
    num_virtual_tokens=100,
    prompt_tuning_init_text="Java code",
    tokenizer_name_or_path=model_name,
)

peft_model = get_peft_model(model, peft_config)
print(peft_model.print_trainable_parameters())
```

Nella configurazione del prompt tuning, è stato impostato il parametro `num_virtual_tokens` a 100, segnalando l'impiego di cento token virtuali. Questi token rappresentano gli elementi modificabili che il modello può perfezionare durante l'addestramento, offrendogli così la capacità di adattarsi senza la necessità di alterare la sua struttura interna complessiva. La presenza di questi token virtuali introduce un livello di flessibilità notevole, pur essendo in numero estremamente limitato rispetto al totale dei parametri del modello. Infatti, i cento token virtuali costituiscono soltanto lo 0.006% dell'intera quantità di parametri del modello.

E' stato elezionato *"Java code"* come testo iniziale per il prompt tuning, con l'intento di dirigere il modello verso l'elaborazione e la generazione di codice Java, pur lasciandogli la libertà di esplorare oltre i confini di un ambito troppo definito. La finalità di questa scelta è di incoraggiare il modello a

imparare autonomamente le specifiche e le strutture tipiche del linguaggio Java, consentendogli di sviluppare le proprie abilità in un modo più spontaneo e adattabile.

Parametri impostati per il training:

```
from transformers import TrainingArguments, Trainer

trainingArgs = TrainingArguments(
    output_dir="./prompt_tuned_model",
    num_train_epochs=20,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=1,
    gradient_checkpointing=True,
    gradient_checkpointing_kwargs={'use_reentrant':False},
    optim="adamw_torch",
    logging_steps=5,
    save_strategy="epoch",
    learning_rate=1e-3,
    weight_decay=0.01,
    group_by_length=True,
    lr_scheduler_type="cosine",
    disable_tqdm=True,
    report_to="wandb",
    seed=42,
)
```

- `output_dir`: La directory `"./prompt_tuned_model"` è stata selezionata per salvare i modelli addestrati e gli output del training;
- `num_train_epochs`: Un numero di 20 epoche è stato scelto per garantire che il modello abbia sufficiente esposizione ai dati di training da apprendere;
- `per_device_train_batch_size`: una dimensione di batch di 4 per l'hardware bilancia l'uso efficiente della memoria e la capacità di generalizzazione, permettendo una convergenza stabile del modello;
- `gradient_accumulation_steps`: Questo parametro è impostato a 1 per applicare gli aggiornamenti del gradiente dopo ogni step, ottimizzando il training senza necessità di accumulare gradienti su più step;
- `gradient_checkpointing`: Abilita il checkpointing dei gradienti per risparmiare memoria;

- `optim`: l'ottimizzatore AdamW fornisce un equilibrio tra efficienza e efficacia nell'aggiornamento dei pesi del modello, con benefici nella regolarizzazione e nella gestione dei tassi di apprendimento;
- `logging_steps`: Registrare i log ogni 5 step fornisce feedback regolari sull'andamento del training senza sovraccaricare di informazioni. La strategia (`save_strategy`) per salvare i checkpoint del modello è impostata su `"epoch"`;
- `learning_rate`: Tasso di apprendimento iniziale per l'ottimizzatore di 0.001 è stato selezionato per promuovere un addestramento stabile e coerente, evitando salti troppo grandi che potrebbero portare a una convergenza subottimale;
- `weight_decay`: L'applicazione di un decadimento del peso di 0.01 aiuta a prevenire l'overfitting regolarizzando la magnitudine;
- `lr_scheduler_type`: L'utilizzo di un programmatore del tasso di apprendimento di tipo cosinusoidale favorisce una diminuzione graduale del learning rate, facilitando una convergenza più fine nella fase finale del training;
- `report_to`: Destinazione dei report di addestramento è stato impostato sulla piattaforma `"wandb"`;
- `seed`: Il seed garantisce la riproducibilità degli esperimenti.

Questi parametri forniscono un controllo dettagliato su come l'addestramento del modello viene eseguito, permettendo di ottimizzare il processo per raggiungere i migliori risultati possibili data la configurazione hardware, le dimensioni del modello e le specifiche del task.

Il modello è stato addestrato utilizzando un training set precedentemente preparato, che include 28 istanze selezionate dal dataset originale. Questo ha permesso di distinguere chiaramente tra le istanze impiegate per l'addestramento e quelle destinate alla valutazione.

Il processo di addestramento viene monitorato con wandb.

Weights & Biases (W&B) è una piattaforma avanzata di gestione degli esperimenti per il monitoraggio, l'analisi e l'ottimizzazione dei progetti di machine learning. Nella tesi, l'uso di W&B ha permesso un monitoraggio in tempo reale delle metriche di addestramento, facilitando l'identificazione e la risoluzione di problemi come l'overfitting. La capacità di confrontare facilmente gli esperimenti ha guidato la selezione delle configurazioni ottimali del modello. Inoltre, la piattaforma ha assicurato la riproducibilità degli esperimenti attraverso il versionamento automatico e la documentazione dettagliata.

Capitolo 5

Risultati

5.1 Presentazione dei Risultati

All'interno del benchmark, i test sono stati categorizzati in due gruppi distinti: obbligatori e opzionali, per gli anni accademici 2019 e 2020. Tale distinzione è stata introdotta dal docente responsabile del corso al fine della valutazione degli esami.

E' stata condotta un'analisi comparativa per determinare il numero di test superati da ciascun modello, considerando dapprima esclusivamente quelli obbligatori e successivamente includendo anche i test opzionali. È importante notare che i test opzionali sono generalmente meno numerosi rispetto a quelli obbligatori e, pertanto, hanno un impatto relativamente minore sul risultato complessivo. Tuttavia, data la loro maggiore complessità, tendono a registrare una percentuale inferiore di successo.

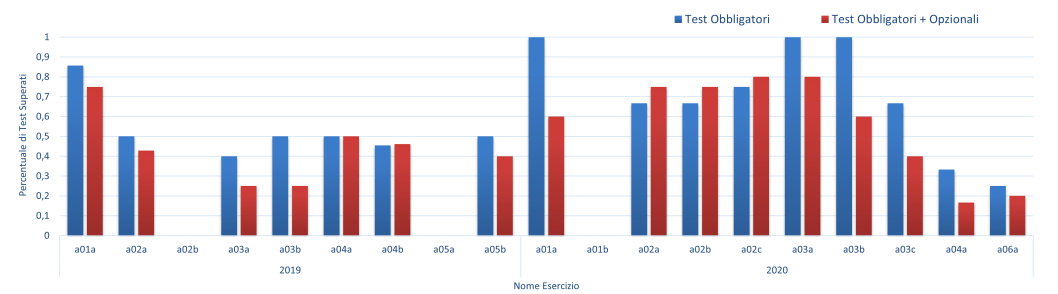

Figura 5.1: Comparazione Percentuale di Test Superati da DeepSeek-Coder, con e senza i Test Opzionali.

Per illustrare l'analisi effettuata, presentiamo un esempio di questo confronto focalizzato sul modello DeepSeek-Coder, i cui risultati sono rappresentati nel Grafico 5.1. Ulteriori dettagli sul numero di test specifici superati vengono

elencati nella Tabella 5.1. Dall'esame dei risultati è emerso che in 11 dei 19 casi analizzati, corrispondenti a circa il 60%, il numero di test superati è maggiore includendo esclusivamente i test obbligatori. In 4 casi, la percentuale di successo è rimasta invariata, mentre nei restanti 4 casi, l'inclusione dei test opzionali ha determinato un incremento della percentuale di test superati.

Confronto tra modelli

Come illustrato nella Tabella 5.2, abbiamo effettuato un'analisi comparativa delle prestazioni di tre Large Language Model avanzati, valutati sugli stessi benchmark. Questa analisi ha incluso sia i test standard che quelli opzionali, coprendo gli esami proposti nel nostro dataset dal 2019 al 2023. Nell'analisi comparativa, sono stati inclusi i risultati ottenuti dai modelli sul benchmark HumanEval [19], un set di dati composto da 164 problemi di programmazione in linguaggio Python, rilasciato da OpenAI.

Si può osservare che le percentuali di test superati nel nostro dataset risultano inferiori rispetto a quelle registrate su HumanEval, evidenziando così una maggiore complessità del nostro benchmark. Questa analisi ha confermato l'esistenza di una differenza in termini di prestazioni tra i modelli CodeLlama e Mistral in confronto a DeepSeek-Coder.

Modello	**Parametri**	**Test superati**	**Test Superati con Opzionali**	**HumanEval**
Deepseek-Coder-6.7-instruct	6.7B	49.5%	44.9%	78.6%
Mistral-7B-Instruct-v0.2	7B	26.4%	22.1%	30.5%
CodeLlama-7b-Instruct-hf	7B	15.5%	13.0%	34.8%

Tabella 5.2: Test Superati.

Il confronto tra i modelli mostra variazioni notevoli nelle loro capacità di superare i test. `DeepSeek-Coder` emerge come il modello più performante, superando il 49.5% dei test standard e il 44.9% dei test opzionali. Al contrario, `Mistral` e `CodeLlama` mostrano prestazioni inferiori, suggerendo limitazioni nella capacità di questi modelli di affrontare problemi complessi e molto strutturati.

Da un'analisi più dettagliata dei risultati, come rappresentato nel Grafico 5.2, emerge un aspetto peculiare riguardante il successo dei modelli nei compiti d'esame. Tra i 42 compiti analizzati, solo 6 sono stati superati completamente da DeepSeek-Coder, 1 da parte di Mistral, mentre CodeLlama non è riuscito a superarne alcuno. Tuttavia, è altrettanto rilevante notare che per 5 di questi esami, nessuno dei tre modelli è riuscito a superare anche solo un singolo test, evidenziando lacune significative nella loro capacità di generalizzare o affrontare determinate tipologie di problemi.

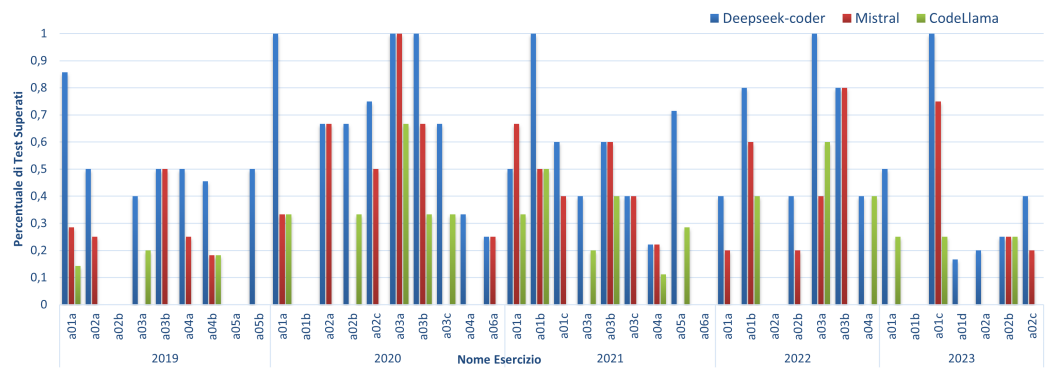

Figura 5.2: Confronto Test Superati fra i Modelli.

Modello con Prompt Tuning vs Senza Prompt Tuning

Durante l'addestramento mediante prompt tuning, l'andamento della funzione di loss, illustrato nel grafico della Figura 5.3, presenta un trend decrescente, sebbene caratterizzato da notevoli fluttuazioni, alternando fasi di calo a momenti di aumento. Questa variabilità può essere attribuita alla diversità e alla complessità delle problematiche affrontate: benché ogni esercizio implichi la generazione di una classe Java, la natura specifica di ciascun compito varia significativamente. Tale variabilità favorisce l'efficacia di determinati prompt in specifici contesti, al punto che, quando il modello si specializza eccessivamente, può essere necessario un aggiustamento per riacquistare una capacità di generalizzazione. Questo fenomeno sottolinea l'importanza di un approccio equilibrato tra specializzazione e generalizzazione nel training del modello.

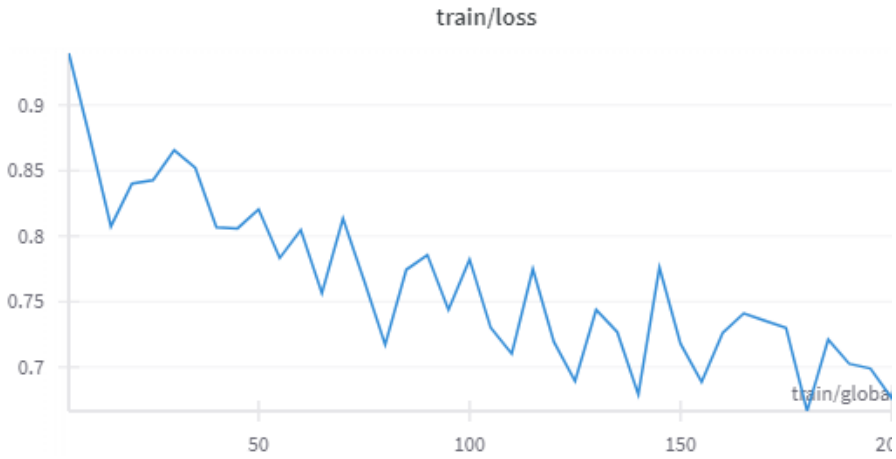

Figura 5.3: Grafico della Funzione di Loss durante l'Addestramento del Prompt.

Per ridurre le fluttuazioni osservate nell'andamento della funzione di loss e favorire un'apprendimento più equilibrato, è stata implementata una strategia di randomizzazione del dataset di training. Questo approccio ha permesso di assicurare una distribuzione bilanciata dei vari tipi di esercizi nel corso dell'addestramento, minimizzando il rischio che il modello si soffermasse eccessivamente su specifiche categorie di problemi.

Inoltre, è stato ulteriormente esplorata la possibilità di stabilizzare l'apprendimento mediante l'adozione di dimensioni di batch più contenute e un learning rate più basso. Questi aggiustamenti, tuttavia, non hanno portato a miglioramenti significativi nelle prestazioni.

Il grafico seguente (cfr. Figura 5.4) mostra un confronto tra i risultati ottenuti sul test set dal modello DeepSeek-Coder prima e dopo il Prompt Tuning.

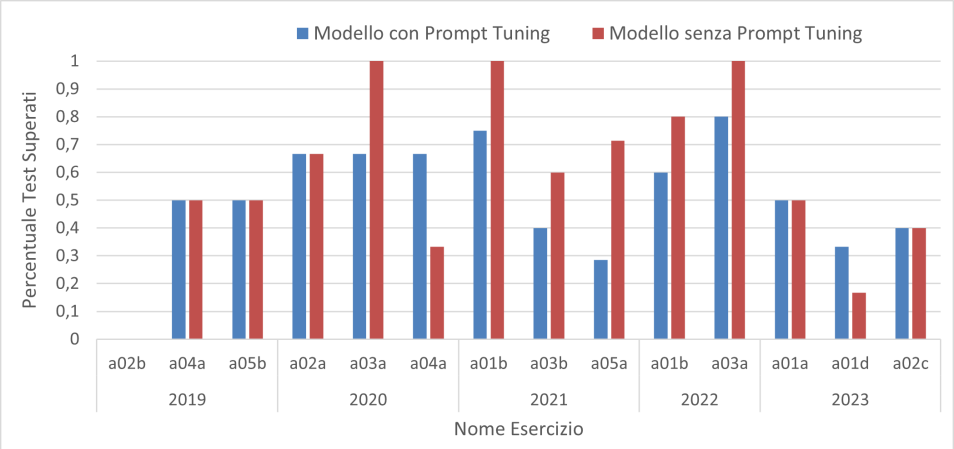

Figura 5.4: Confronto delle Prestazioni: Modello con Prompt Tuning vs Senza Prompt Tuning sul Test Set.

L'analisi del grafico rivela che il modello sottoposto a prompt tuning registra prestazioni leggermente inferiori, con una percentuale di successo nei test del 50.5%, in confronto al modello originale che raggiunge il 58.4% di test superati. Quest'ultimo, infatti, si dimostra superiore in 6 dei 14 confronti e eguaglia le prestazioni del modello addestrato negli altri 6 casi. È rilevante osservare, però, che vi sono 2 situazioni specifiche in cui il modello originale fallisce alcuni test, mentre la variante con prompt tuning riesce a produrre un numero maggiore di risposte corrette.

5.2 Analisi dei Risultati

L'esame dei risultati ottenuti dipinge una panoramica generalmente favorevole, evidenziando l'efficacia dei modelli nell'affrontare sfide di programmazione non banali. Tradizionalmente orientati alla generazione di metodi singoli e ben definiti, i modelli sono stati testati in un contesto più complesso: quello della creazione di intere classi Java. Queste classi dovevano non solo includere diversi metodi correlati tra loro ma anche incorporare l'utilizzo di pattern consolidati come Factory e Helpers. Un elemento chiave della sfida era l'impiego di specifiche interfacce funzionali di Java – quali Function, BiFunction, Predicate, BiPredicate, e UnaryOperator – che rappresentano strumenti avanzati per la gestione di logiche complesse e operazioni su dati. Le soluzioni ottimali, simili a quelle proposte dal professore, tendono inoltre a favorire l'uso degli stream, ulteriormente sottolineando l'orientamento verso pratiche di programmazione moderne e avanzate.

È stato osservato che tutti e tre i modelli hanno dimostrato la capacità di generare classi Java corrette e ben strutturate, complete dell'implementazione dell'interfaccia richiesta e degli import necessari, evitando errori di sintassi. Tuttavia, data la loro formazione su esempi di classi più semplici, tendono a trattare ogni metodo individualmente, senza una visione globale del problema.

DeepSeek-Coder si è dimostrato particolarmente efficace nella generazione di classi Java compilabili nella quasi totalità dei casi, segno di una notevole comprensione del compito assegnato e della capacità di integrare metodi in modo coerente. Tuttavia, nonostante questa abilità, il codice prodotto spesso non rispecchia i principi della OOP, principalmente a causa di una tendenza alla ripetizione, abbassandone la qualità. Questo si riflette anche in un utilizzo relativamente scarso degli stream.

Al contrario, CodeLlama e, in particolare, Mistral, hanno dimostrato una forte inclinazione verso l'utilizzo degli stream in Java, adottando spesso approcci complessi e strutturati. Tuttavia, questa scelta ha a volte causato la generazione di codice non compilabile, dovuto a incongruenze nei tipi di oggetti processati. Il problema principale è la creazione e concatenazione di blocchi nello stesso stream che elaborano tipi di oggetti non conformi ai dati attesi, portando a errori di compilazione. Un esempio di errore:

Type mismatch: cannot convert from IntStream to Stream<Pair<X,X»

Una nota positiva riconosciuta è stato l'efficace lavoro con la libreria java.utils.stream.Collections, dimostrando una buona comprensione nel costruire soluzioni adeguate al tipo di dato restituito dal metodo.

Metodo corretto implementato da DeepSeek-Coder:

```
@Override
public <E> Iterator<Iterator<E>> increasingSizeWithSingleton(E e) {
    return new Iterator<>() {
        private int size = 0;

        @Override
        public boolean hasNext() {
            return true;
        }

        @Override
        public Iterator<E> next() {
            size++;
            final int currentSize = size;
            return new Iterator<>() {
                private int index = 0;

                @Override
                public boolean hasNext() {
                    return index < currentSize;
                }

                @Override
                public E next() {
                    if (index >= currentSize) throw new NoSuchElementException();
                    index++;
                    return e;
                }
            };
        }
    };
}
```

Metodo non corretto implementato da CodeLlama (errore sulla flatMap):

```
@Override
public <E> Iterator<Iterator<E>> increasingSizeWithSingleton(E element) {
    return Stream.iterate(1, i -> i + 1)
            .map(i -> Stream.iterate(i, j -> j + 1)
                    .map(j -> element).limit(i)
                    .iterator()).flatMap(Function.identity())
                    .iterator();
}
```

CodeLlama, inoltre, ha registrato prestazioni inferiori a causa di una frequente incapacità di comprendere pienamente il compito richiesto, avendo come risultato metodi correttamente implementati che però falliscono i test.

Si è notato che di fronte a classi particolarmente complesse o estese, i modelli adottano strategie diverse: DeepSeek-Coder tende a scrivere un primo metodo completo lasciando un commento di implementare i metodi successivi in modo similare, mentre CodeLlama e Mistral prediligono delineare la struttura della classe, lasciando i metodi con un semplice commento dove andrà collocata l'implementazione.

```
import java.util.List;
import java.util.Set;

public class WorkflowsFactoryImpl implements WorkflowsFactory {

    @Override
    public <T> Workflow<T> singleTask(T task) {
        // Complete the implementation of the method here
    }

    @Override
    public <T> Workflow<T> tasksSequence(List<T> tasks) {
        // Complete the implementation of the method here
    }

    ...
}
```

Un fattore cruciale è l'omogeneità dei ruoli richiesti dai metodi dell'interfaccia: quando questi sono simili, i modelli mostrano una migliore comprensione complessiva e generano soluzioni più coerenti. Invece, la diversità dei compiti tra i metodi può confondere i modelli, compromettendo la qualità anche delle soluzioni a metodi più semplici.

Infine, è stato evidenziato un maggiore tasso di successo nella generazione di classi dedicate a compiti astratti, come la creazione di alberi o la manipolazione di liste. Al contrario, i compiti che simulano scenari reali come la gestione di banche, poste o università risultano meno compresi, con una conseguente riduzione della probabilità di superare i test.

Modello con Prompt Tuning

I risultati ottenuti suggeriscono che le fluttuazioni riscontrate durante l'addestramento del prompt potrebbero derivare da fattori intrinseci alla natura dei dati o alla complessità dei compiti da apprendere, piuttosto che dai parametri di addestramento. In quanto anche modificando questi parametri la curva della funzione di loss (cfr. Figura 5.3) rimane con un andamento molto similare.

Analizzando il numero di test superati, si osserva che il modello implementato con il prompt tuning registra una riduzione di circa '8% nella percentuale di successo rispetto al modello originale. Tuttavia, emerge che questo stesso modello è capace di affrontare e risolvere quesiti che precedentemente, con prompt manualmente generati, rimanevano irrisolti. Questo fenomeno indica che, nonostante un leggero calo nelle prestazioni generali, il prompt tuning fornisce il modello di una versatilità superiore nell'elaborare e superare sfide precedentemente inaccessibili, evidenziando un miglioramento nella capacità di adattamento a problemi complessi.

5.3 Esempio di Input - Output

Input fornito al modello:

```
You are an accomplished Java developer tasked with creating complex, generic data structures and ensuring
    they meet rigorous interface specifications.

@@Instruction

Your job is to create a Java class named 'ScannerFactoryImpl' that implements the 'ScannerFactory'
    interface. Use the following interface:

public interface Scanner<X,Y>{
    Y scan(Iterator<X> input);
}

public interface ScannerFactory {

    /**
     * Creates a Scanner that, given a sequence, filters and maps its elements.
     *
     * @param <X> the type of input's elements
     * @param <Y> the type of output's elements
     * @param filter A Predicate that selects elements to be included based on a condition
         (java.util.function.Predicate).
     * @param mapper A Function that transforms elements of type X to type Y
         (java.util.function.Function).
     * @return A Scanner that returns a list of elements of type Y, which are the result of applying
     *          the mapper to elements of type X that satisfy the filter condition.
     *          For example, given the sequence (10,20,30,40) with a filter x->x>20 and a mapper
        x->x*2,
     *          it returns (60,80).
     */
    <X,Y> Scanner<X,List<Y>> collect(Predicate<X> filter, Function<X,Y> mapper);

    /**
     * Creates a Scanner that, given a sequence, finds the first element that satisfies a given
         condition.
     *
     * @param <X> the type of input's elements, and of the single, optional output
     * @param filter A Predicate that tests elements.
     * @return A Scanner that returns an Optional describing the first element of type X that satisfies
     *          the filter condition, or an empty Optional if no such element is found.
     *          For example, given the sequence (10,20,31,41) with a filter x->x%2==1, it returns
        Optional.of(31).
     */
    <X> Scanner<X,Optional<X>> findFirst(Predicate<X> filter);

    /**
     * Creates a Scanner that finds the maximum element of the maximal prefix ordered subsequence in
         a sequence of integers.
     *
     * @return A Scanner that returns an Optional<Integer> describing the maximum element of the
         maximal prefix
     *          ordered subsequence, or an empty Optional (java.util.Optional) if the input list is
        empty.
     *          For example, given the sequence (10,20,31,5,60), it returns Optional.of(31) because the
        maximal
     *          prefix ordered subsequence is (10,20,31).
     */
    Scanner<Integer,Optional<Integer>> maximalPrefix();
```

```
    /**
     * Creates a Scanner that computes cumulative sums of a sequence of integers.
     *
     * @return A Scanner that returns a list of cumulative sums. For example, given the sequence
     *           (10,20,30,40), it returns (10,30,60,100).
     *           This method is marked as optional and may not be required in all implementations or
          exams.
     */
    Scanner<Integer,List<Integer>> cumulativeSum();

}

Only if necessary use:
public interface Pair<X,Y> {
  X getX();
  Y getY();
  int hashCode();
  boolean equals(Object obj);
}

In response, implement all methods of the ScannerFactory interface. When possible, use Java patterns and
     compact code with streams.

@@Response
```

Output generato dal modello:

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;
import java.util.Optional;
import java.util.Spliterator;
import java.util.Spliterators;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.StreamSupport;

public class ScannerFactoryImpl implements ScannerFactory {

    @Override
    public <X,Y> Scanner<X,List<Y>> collect(Predicate<X> filter, Function<X,Y> mapper) {
        return new Scanner<X,List<Y>>() {
            @Override
            public List<Y> scan(Iterator<X> input) {
                List<Y> result = new ArrayList<>();
                while(input.hasNext()) {
                    X next = input.next();
                    if(filter.test(next)) {
                        result.add(mapper.apply(next));
                    }
                }
                return result;
            }
        };
    }

    @Override
    public <X> Scanner<X, Optional<X>> findFirst(Predicate<X> filter) {
        return new Scanner<X, Optional<X>>() {
            @Override
            public Optional<X> scan(Iterator<X> input) {
```

```
                return StreamSupport.stream(Spliterators.spliteratorUnknownSize(input,
                        Spliterator.ORDERED), false)
                            . filter ( filter )
                            . findFirst ();
            }
        };
    }

    @Override
    public Scanner<Integer, Optional<Integer>> maximalPrefix() {
        return new Scanner<Integer, Optional<Integer>>() {
            @Override
            public Optional<Integer> scan(Iterator<Integer> input) {
                if (!input.hasNext()) {
                    return Optional.empty();
                }

                List<Integer> list = new ArrayList<>();
                while (input.hasNext()) {
                    list .add(input.next());
                }

                return Optional.of( list .stream()
                            .map(i -> new Pair<>(i, list.subList(0, list .indexOf(i) + 1)))
                            .max(Comparator.comparing(p ->
                                    p.getY().stream().max(Integer::compareTo).orElse(0)))
                            .map(p -> p.getY().stream().max(Integer::compareTo).orElse(0))
                            .orElse(0));
            }
        };
    }

    @Override
    public Scanner<Integer, List<Integer>> cumulativeSum() {
        return new Scanner<Integer, List<Integer>>() {
            @Override
            public List<Integer> scan(Iterator<Integer> input) {
                List<Integer> result = new ArrayList<>();
                int sum = 0;
                while (input.hasNext()) {
                    sum += input.next();
                    result .add(sum);
                }
                return result ;
            }
        };
    }
}
```

Anno	Esercizio	Nome Interfaccie	Test Superati	Con Opzionali
2019	a01a	GraphFactory	6/7	6/8
2019	a02a	WorkflowsFactory	2/4	3/7
2019	a02b	ExamsFactory	0/2	0/4
2019	a03a	RWControllersFactory	2/5	2/8
2019	a03b	PostOfficeFactory	1/2	1/4
2019	a04a	SequencesProvidersFactory	2/4	3/6
2019	a04b	ParsersFactory	5/11	6/13
2019	a05a	IteratorOfListsFactory	0/3	0/5
2019	a05b	IteratorIteratorFactory	2/4	2/5
2020	a01a	TreeFactory	3/3	3/5
2020	a01b	MathematicalFunctionsFactory	0/6	0/8
2020	a02a	ScannerFactory	2/3	3/4
2020	a02b	PatternExtractorFactory	2/3	3/4
2020	a02c	ListSplitterFactory	3/4	4/5
2020	a03a	GroupingFactory	3/3	4/5
2020	a03b	EquivalenceFactory	3/3	3/5
2020	a03c	TableFactory	2/3	2/5
2020	a04a	FunctionalStreamFactory	1/3	1/6
2020	a06a	BTreeFactory	1/4	1/5
2021	a01a	AcceptorFactory	3/6	
2021	a01b	EventSequenceProducerHelpers	4/4	
2021	a01c	EventHistoryFactory	3/5	
2021	a03a	DecisionChainFactory	2/5	
2021	a03b	LensFactory	3/5	
2021	a03c	DeterministicParserFactory	2/5	
2021	a04a	EitherFactory	2/9	
2021	a05a	StateFactory	5/7	
2021	a06a	CirclerFactory	0/4	
2022	a01a	SubsequenceCombinerFactory	2/5	
2022	a01b	FlattenerFactory	4/5	
2022	a02a	RecursiveIteratorHelpers	0/5	
2022	a02b	CursorHelpers	2/5	
2022	a03a	ParserFactory	5/5	
2022	a03b	LazyTreeFactory	4/5	
2022	a04a	BankAccountFactory	2/5	
2023	a01a	TimetableFactory	2/4	
2023	a01b	TimeSheetFactory	0/4	
2023	a01c	TimeSheetFactory	4/4	
2023	a01d	TimetableFactory	1/6	
2023	a02a	ListBuilderFactory	1/5	
2023	a02b	RulesEngineFactory	1/4	
2023	a02c	ReplacersFactory	2/5	

Tabella 5.1: Test Superati da DeepSeek-Coder-6.7-Instruct.

Conclusioni e Sviluppi Futuri

Il presente studio si è concentrato sulla valutazione delle capacità degli attuali modelli di generazione di codice open source, mettendoli alla prova con compiti avanzati di programmazione Java caratteristici del contesto accademico universitario, richiesti solitamente a studenti laureandi. L'obiettivo era esaminare la loro efficacia nell'implementare classi Java complesse, includendo un'ampia varietà di metodi e l'adozione di pattern consolidati oltre a interfacce funzionali avanzate. Il dataset ricavato da questi esercizi di programmazione è diventato una risorsa preziosa per la comunità scientifica, introducendo un nuovo standard per valutare le prestazioni dei modelli di intelligenza artificiale nella generazione di codice.

Nel corso di questo studio, tre Large Language Model - Deepseek-Coder, Mistral e CodeLlama - sono stati messi alla prova sul nuovo benchmark. Questi modelli sono stati selezionati in base ai loro risultati precedentemente documentati in letteratura e al loro numero contenuto di parametri, circa 7 miliardi. I risultati hanno evidenziato che, nonostante una performance generalmente positiva, DeepSeek-Coder si è distinto superando circa il 50% dei test proposti, completando integralmente sei compiti d'esame. Tuttavia, emerge chiaramente che c'è ancora ampio margine di miglioramento.

Una delle sfide principali incontrate riguarda la generazione di codice conforme ai principi della programmazione orientata agli oggetti. I modelli hanno mostrato difficoltà nell'approcciare la creazione di classi in modo globale, limitandosi spesso alla realizzazione di singoli metodi e non sfruttando appieno i pattern di Java per costruzioni più elaborate. Questo ha portato a un uso ridondante del codice, compromettendo l'efficienza.

È stato inoltre osservato che i modelli faticano particolarmente nell'elaborare e generare codice che utilizza efficacemente gli stream di Java, a causa della complessità intrinseca delle linee di codice richieste e della necessità di una profonda comprensione del contesto e delle specifiche necessità di elaborazione dei dati. La complessità delle richieste ha talvolta portato alla creazione di codice non compilabile, sebbene in una percentuale limitata rispetto al totale, sottolineando l'importanza di una valutazione manuale accurata.

Lo studio ha, inoltre, evidenziato l'importanza del prompt tuning nell'incre-

mentare la versatilità dei modelli, migliorando la loro capacità di affrontare e risolvere problemi precedentemente fuori portata. Questo processo non solo amplia la gamma di sfide che i modelli possono superare ma evidenzia anche l'evoluzione continua delle strategie volte a ottimizzare le prestazioni di tali sistemi.

Guardando al futuro, emerge l'interesse per l'esplorazione di tecniche di prompt tuning più sofisticate, nonché sulla possibilità di utilizzare l'inversione dell'embedding post-addestramento per decodificare i prompt in linguaggio naturale comprensibile dall'uomo. Questo approccio apre nuove frontiere nella ricerca, puntando a decifrare più a fondo i meccanismi attraverso cui il prompt tuning contribuisce all'elevazione delle capacità generative dei modelli.

Un'osservazione fondamentale è che l'efficacia di un modello nella generazione di codice non dipende tanto dalla quantità dei suoi parametri, quanto dalla sua architettura e dai dati su cui viene addestrato. L'analisi ha dimostrato che modelli con un numero di parametri simile possono produrre risultati diversi, evidenziando l'importanza di una selezione accurata e dell'ottimizzazione dei dataset per potenziare le capacità dei modelli.

Questo lavoro conferma il potenziale degli strumenti di intelligenza artificiale nel semplificare lo sviluppo software e suggerisce direzioni per ulteriori miglioramenti. La sfida principale rimane quella di ottimizzare i modelli per una comprensione e applicazione efficaci dei principi della programmazione orientata agli oggetti, generando codice non solo corretto sintatticamente, ma anche efficiente e in linea con le *best practice* di programmazione.

In conclusione, l'applicazione di Large Language Model alla generazione di codice rappresenta un campo di ricerca dinamica e in rapida espansione. I progressi ottenuti finora aprono nuove possibilità per semplificare la scrittura di codice complesso e per esplorare metodologie innovative per una programmazione assistita, rendendo il processo di sviluppo software più efficiente, accurato e accessibile. L'evoluzione continua e l'affinamento delle tecniche di intelligenza artificiale mantengono questo ambito di ricerca estremamente attuale.

Ringraziamenti

Il primo ringraziamento va al relatore di questo lavoro, il Prof. Gianluca Moro, che ha reso possibile tutto ciò e ha acceso il mio personale interesse nei confronti di questa splendida disciplina. La sua capacità di stimolare il pensiero critico e di incoraggiare l'esplorazione di nuove idee ha arricchito la mia esperienza accademica.

Un ringraziamento speciale va anche ai tutor, Giacomo Frisoni e Luca Ragazzi, per il loro sostegno e supporto nello svolgimento della tesi. La loro disponibilità, competenza e capacità di fornire consigli preziosi hanno contribuito in modo significativo al mio percorso di ricerca e apprendimento.

Grazie profondamente ai miei genitori e alla mia famiglia per avermi dato l'opportunità di studiare e per tutto il sostegno incondizionato fornito durante questi anni. Il loro amore e incoraggiamento sono stati essenziali in ogni fase del mio percorso, offrendomi una base solida su cui appoggiarmi nei momenti di difficoltà. La loro costante fiducia nelle mie capacità mi ha motivato a impegnarmi con determinazione verso i miei obiettivi.

Infine, ma non meno importante, desidero ringraziare i miei amici e compagni di università che mi sono stati accanto dal primo giorno. La loro presenza, sia durante le lezioni che negli esami, e il loro contributo ai progetti che sono stati realizzati, hanno arricchito notevolmente la mia esperienza universitaria. Il senso di comunità e il sostegno reciproco che abbiamo condiviso sono stati fondamentali per superare le sfide accademiche e personali.

A tutti voi, il mio più sincero ringraziamento per aver reso questo percorso non solo possibile, ma anche straordinariamente significativo.

Bibliografia

[1] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196, 2024.

[2] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. *CoRR*, abs/2310.06825, 2023.

[3] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[5] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *CoRR*, abs/2307.06435, 2023.

[6] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *CoRR*, abs/2104.08691, 2021.

[7] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *CoRR*, abs/2101.00190, 2021.

[8] Haoran Li, Mingshi Xu, and Yangqiu Song. Sentence embedding leaks more information than you expect: Generative embedding inversion attack to recover the whole sentence. *CoRR*, abs/2305.03010, 2023.

[9] John X. Morris, Volodymyr Kuleshov, Vitaly Shmatikov, and Alexander M. Rush. Text embeddings reveal (almost) as much as text. *CoRR*, abs/2310.06816, 2023.

[10] Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *CoRR*, abs/2401.08500, 2024.

[11] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.

[12] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G., Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H. Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions. *CoRR*, abs/2305.10435, 2023.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[14] Tong Xiao and Jingbo Zhu. Introduction to transformers: an NLP perspective. *CoRR*, abs/2311.17633, 2023.

[15] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *CoRR*, abs/1803.11175, 2018.

[16] Pu-Chin Chen, Henry Tsai, Srinadh Bhojanapalli, Hyung Won Chung, Yin-Wen Chang, and Chun-Sung Ferng. A simple and effective positional encoding for transformers. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP*

2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, pages 2974–2988. Association for Computational Linguistics, 2021.

[17] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[18] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[20] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. A survey on language models for code. *CoRR*, abs/2311.07989, 2023.

[21] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. Large language models meet nl2code: A survey. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 7443–7464. Association for Computational Linguistics, 2023.

[22] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey

of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9):195:1–195:35, 2023.

[23] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *CoRR*, abs/2303.07839, 2023.

[24] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *CoRR*, abs/2302.11382, 2023.

[25] Sondos Mahmoud Bsharat, Aidar Myrzakhan, and Zhiqiang Shen. Principled instructions are all you need for questioning llama-1/2, GPT-3.5/4. *CoRR*, abs/2312.16171, 2023.

[26] Biqiang Mu, Tianshi Chen, He Kong, Bo Jiang, Lei Wang, and Junfeng Wu. On embeddings and inverse embeddings of input design for regularized system identification. *CoRR*, abs/2209.13152, 2022.

[27] Asma Ghandeharioun, Avi Caciularu, Adam Pearce, Lucas Dixon, and Mor Geva. Patchscopes: A unifying framework for inspecting hidden representations of language models. *CoRR*, abs/2401.06102, 2024.

[28] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[29] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[30] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022.