Large Language Model: Estrazione Dati Strutturati da File non Strutturati

Daniel Acampora, Università degli Studi di Brescia

Tirocinio Curriculare 12 CFU

presso Garda Informatica Snc di Lorenzo e Giovanni Chiodi

Indice

Informazioni generali su Large Language Model

Un Large Language Model (LLM) è un modello di intelligenza artificiale (IA) volta al Natural Language Processing (NLP) che si suddivide in Natural Language Understanding (NLU) e Natural Language Generation (NLG). Quindi un LLM è un IA in grado di comprendere il linguaggio umano e di generare testo come un umano. Il Transformer è la tecnologia alla base dei LLM, il quale è a sua volta l'evoluzione delle Reti Neurali. Infatti, il Transformer è composto da più strati (layers) di reti neurali.

In generale i Language Model sono modelli stocastici generativi perché generano testo sulla base di calcoli probabilistici. In particolare, viene fatta la tokenization dell'input cioè la suddivisione di un testo in token della stessa dimensione. I token vengono usati per creare una rappresentazione numerica di unità di testo. Poi i modelli sulla base dell'input e dei propri parametri prevedono qual è il token più adeguato da dare in output anche sulla base di quelli già prodotti. Ciascun modello ha un tokenizer con un proprio metodo di tokenization, ma in media un token equivale a circa quattro caratteri. A volte token vettorizzati vengono chiamati embeddings.

Le Recurrant Neural Network (RNN) usano un metodo sequenziale auto regressivo per generare l'output in base all'input e all'output precedentemente prodotto. Questa natura sequenziale limitava la parallelizzazione. Invece, il Transformer con l'utilizzo del meccanismo dell'attenzione può parallelizzare i calcoli, così ha potuto sfruttare al meglio le risorse di calcolo come GPU e TPU. I modelli successivi al Transformer hanno avuto pochi cambiamenti architetturali, il più significativo è la preferenza dell'architettura decoder-only a quella encoder-decoder.

Sulla base del Transformer sono stati creati i Pre-trained Language Model, addestrati su task specifici come la traduzione o il riassunto. I PLM sono diventati LLM perché con l'aumentare dei parametri dei modelli si è notato un aumento delle loro capacità divenendo adatti all'uso general purpose, ad esempio riassunti, traduzione, complemento di testo, recupero informazioni e analisi dei sentimenti. In genere si parla di LLM da circa 500M di parametri in su.

Però l'aumentare dei parametri causa problemi di requisiti minimi di risorse di calcolo sempre più costose e problemi di qualità dei dati usati per l'addestramento che in caso di cattiva qualità possono portare a generazione di testo inutile o addirittura malevolo.

Perciò si è iniziato a porre l'accento anche su modelli lightweight, creati con dataset di alta qualità.

In più i LLM sono ottimi per casi d'uso generici ma quando si vuole ottenere risultati relativi a uno specifico ambito può essere necessaria la pratica del fine-tuning. Ciò può richiedere ulteriori risorse di calcolo e di dati perché per fare fine-tuning bisogna curare uno o più dataset su cui fare l'addestramento aggiuntivo.

Ciclo di vita di un LLM

Lo sviluppo di un LLM si suddivide in più fasi, le principali sono, la raccolta di dati, il pre-training, il fine tuning, la valutazione e l'inferenza.

1 – Raccolta di dati

La raccolta di dati (o data preparation) è forse la più delicata perché per il pre-training vengono raccolti enormi moli di dati, anche nell'ordine dei trilioni. Questi dati sono la base del funzionamento e del comportamento del modello. I metodi di raccolta e di preparazione variano da organizzazione a organizzazione, soprattutto riguardo ai dati presi dal web che sono i più potenzialmente pericolosi, ma più facili da reperire. Ad esempio, OpenAI con le prime versioni di GPT (Generative Pre-trained Transformer), aveva problemi di generazione di testi offensivi perché una grossa porzione di dati era stata presa dai social network senza alcun controllo sulla qualità dei contenuti.

2 – Pre-training

Durante il pre-training un modello encoder-decoder o decoder-only viene allenato su un enorme insieme di dati che andranno a costituire i suoi parametri. I parametri di un modello pre-trained sono un insieme di costanti chiamate weights and biases che vanno a definire come si propagano i dati del modello. In parole povere un parametro viene considerato un neurone di una delle reti neurali di cui è composto il modello. I parametri vengono rappresentati con la struttura algebrica del tensore che è un array multidimensionale.

Per riassumere, i dati dopo essere stati raccolti e preparati vengono tokenizzati. Il modello, che alla base è una funzione stocastica, viene addestrato sui token dai quali vengono generati i parametri.

3 – Fine-tuning

Dopo il pre-training un modello è già utilizzabile per la generazione di testo, ma è poco affidabile perché ha maggiore probabilità di allucinazioni e di generare testo infinto o inutile. Perciò viene fatto il fine-tuning che è un ulteriore addestramento, con l'obiettivo di istruire il modello a seguire istruzioni e a rispondere nei modi preferiti dagli utenti. Non a caso le organizzazioni rilasciano anche una versione **instruct**, ovvero un modello già fine-tuned per l'esecuzione di istruzioni. Ci sono diverse tecniche di fine-tuning alcune più efficaci ma poco efficienti a livello di requisiti hardware ed altre potenzialmente dannose per il modello. Per esempio, il full fine-tuning riscrive i weights, pratica molto costosa che ha il rischio di rovinare il funzionamento del modello, ma se fatto bene crea un modello completamente riadattato per lo scopo desiderato. Mentre un fine-tuning a basso costo è il Low-Rank Adaptation fine-tuning. Il LoRA fine-tuning scrive una matrice di weights chiamata adapter che viene aggiunta alle matrici originali di un modello, così il funzionamento di base del modello non rischia peggioramenti e nella maggior parte dei casi c'è un leggero miglioramento nelle prestazioni del modello.

4 – Valutazione

I modelli vengono valutati dalle organizzazioni usando dei benchmark per misurare l'accuratezza in diversi task, ad esempio: MMLU per il multitasking, GPQA per le capacità di ragionamento, HumanEval per la codifica in python, MATH per problemi matematici, BFCL per l'abilità della chiamata di funzioni e tool, MGSM per le capacità multilinguistiche. Le valutazioni su benchmark di questo tipo sono utili per fare un primo confronto tra modelli ma non per capire quanto un modello sia adatto a svolgere una specifica task.

5 – Inferenza

L’inferenza è un ragionamento deduttivo. Riguardo agli LLM è la fase di utilizzo del modello. Quindi, un utente dà un prompt in pasto a un modello che dedurrà la sua risposta partendo dal prompt e dai propri parametri. I layer di parametri del modello vengono di volta in volta caricati e rimossi dalla memoria per l’elaborazione della risposta. L’output calcolato da un layer viene usato da quello successivo fino a che non è stato generato l’ultimo token della risposta.

Gli utenti a seconda dell’ambiente che stanno usando per l’inferenza possono controllare più o meno impostazioni e variabili. La più importante da avere presente è la context length o context window. La context length è la somma dei token di input e di output. Se un modello va oltre la sua context length massima darà problemi di dimenticanza e passaggi illogici fino ad arrivare all’allucinazione. Una context length di grosse dimensioni può causare peggioramenti alle prestazioni e all’accuratezza del modello.

In questa fase spesso si parla di prompting o di prompt engineering, che è un insieme di tecniche per comunicare efficacemente query e istruzioni ai modelli. É importante soprattutto sapere la differenza tra zero-shot learning (o zero-shot prompting) e few-shot learning (a volte chiamato in-context learning). Con zero-shot learning si parla della capacità di un modello di rispondere correttamente a un prompt senza avere avuto alcun addestramento o esempio sulla richiesta fatta. Con few-shot learning, invece, si intende la capacità di un modello di rispondere correttamente dopo uno o più esempi. I LLM sono considerati buoni zero-shot learner, a differenza di altri LM, ma solo riguardo a task semplici. Con task complesse è necessario il few-shot learning che in genere da ottimi risultati rendendo superfluo il fine-tuning. Però il few-shot learning essendo fatto durante l’inferenza occupa una porzione di context length, perciò ci possono essere dei peggioramenti nella velocità di risposta del modello.

A seconda dell’applicazione o degli strumenti che si stanno usando per fare inferenza cambiano le modalità per il few-shot learning, ma la base resta sempre la stessa. La prima parte del prompt si chiama System, dove viene spiegato al modello il task, come si deve esprimere e altri vincoli che deve rispettare. Poi c’è la parte degli esempi dove vengono scritti uno o più esempi di coppie di query dell’utente e risposta del modello. Così il modello prima di dover rispondere a delle nuove query ha nel suo contesto delle istruzioni e degli esempi da seguire. Alcuni strumenti permettono di scrivere dei

file con specificati System, esempi ed alcuni parametri come la context length massima. Questi possono essere usati automaticamente quando si usa un modello.

Quantizzazione

Un modello più parametri ha, più sono le sue capacità di elaborazione dei dati, ma aumentano le risorse di calcolo necessarie. Ad esempio, un modello da 7B di parametri viene caricato a malapena in 8GB di VRAM. Questo dipende dalla precisione dei weights dei parametri. I modelli sono addestrati con precisione float 32 (FP32), ciò vuol dire che un parametro è salvato con 32 bit di memoria. Mentre i modelli rilasciati al pubblico di solito hanno mezza precisione, FP16. Quindi, un modello da 7B per ogni parametro deve usare 2 byte di memoria.

Per risolvere questo problema di costosi requisiti di calcolo sono state sviluppate alcune tecniche, tra le quali c'è la quantizzazione, simile alla compressione dati. Grazie alle tecniche di quantizzazione la precisone dei weights può essere ridotta addirittura fino a INT4, riducendo la memoria necessaria e rendendo addirittura possibile un maggiore uso della cache. Perciò, con la quantizzazione si ottengono modelli con tempi di risposta in inferenza più rapidi. Ma la riduzione di precisione dei weights può causare un peggioramento dell'accuratezza del modello. Un modello viene addestrato con quantizzazione FP32, quindi questo è il 100% della sua accuratezza. Il modello, quando viene quantizzato a un livello inferiore, ha una perdita di accuratezza. A FP16 l'accuratezza è di circa il 99%, una perdita quasi trascurabile. A INT8 la precisione è di circa 95%. A INT4 scende introno all'85%. La quantizzazione a INT8 è un ottimo compromesso, se si vuole risparmiare sulle risorse di sistema o velocizzare i tempi di risposta del modello.

Per aggirare il problema dei requisiti hardware alcuni ambienti di utilizzo dei LLM, come llama.cpp e Ollama, permettono di usare sia la GPU che la CPU con le realtive memorie. Questo può portare a peggioramenti nell'accuratezza e nei tempi di risposta.

Requisiti di Sistema

Per i LLM in locale è preferibile l'uso di sistemi operativi basati su Linux, per i quali le librerie Python più usate per il deep learning sono sempre supportate, ad esempio pyTorch o TensorFlow. Per quanto riguarda l'hardware, all'aumentare della quantità dei parametri di un modello aumentano i requisiti. Fino ai 13B di parametri i componenti di consumo sono sufficienti. La GPU è il componente principale poiché deve eseguire i calcoli matriciali dei tensori dei layer dei modelli. Sono preferibili le schede grafiche NVIDIA grazie ai CUDA che sono sempre supportati dai software dedicati all'IA generativa e al deep learning. Riguardo alle specifiche bisogna prestare attenzione alla VRAM. Tenendo conto della quantizzazione FP16, per i modelli lightweight fino a 3B 8 GB di VRAM sono più che sufficienti, per 8B servono 16 GB di VRAM. Però, grazie alla quantizzazione questi requisiti possono essere ridotti. Ad esempio, un modello 3B a INT8 necessita di meno di 4GB di VRAM, un 8B di circa 8 GB. I requisiti hardware dipendono anche dai software usati per l'inferenza. Usando llama.cpp per l'inferenza, il carico del modello viene distribuito dinamicamente tra CPU e GPU, quindi tra RAM e VRAM. Però i continui swap di memoria causano rallentamenti, quindi è consigliabile non fare troppo affidamento su questo se si vogliono tempi di risposta rapidi con task complessi. Comunque, il sistema deve essere dotato di CPU e RAM adeguati alla GPU.

Si consiglia la visione di questo video che parla di requisiti di sistema in generale e la consultazione di questo articolo che parla dei requisiti di sistema dei modelli Llama 3.

LLM per Estrazione Dati

L'estrazione dati è un task molto complesso per i LLM, soprattutto nel caso di estrarre dati strutturati da documenti non strutturati. Con lo zero-shot learning quasi nessun modello è in grado di comprendere il task. È necessario lavorare di prompt engineering, creando almeno un prompt di sistema che descriva il task. Infatti, ChatGPT riesce a estrarre dati col corretto formato solo dopo un'esaustiva descrizione dell'obiettivo. Con few-shot learning, modelli da 7B e 8B hanno risultati molto simili a quelli di ChatGPT con GPT-4o. Anche i modelli 3B riescono a volte a dare dei buoni risultati. Per quanto riguarda la formattazione dell'output, con il solo prompt engineering i risultati non sono sempre consistenti e ci sono problemi di formattazione.

Fortunatamente, gli ambienti di inferenza, come Ollama o LangChain, forniscono tool per costringere il modello a seguire un formato per l'output. Questo, però causa dei peggioramenti nelle prestazioni e nella qualità della risposta.

I LLM sono in grado di riconoscere i dati e di distinguerli, ma non sono sempre capaci di raggrupparli. Per esempio, possono estrare dati comuni come indirizzi, prezzi e dimensioni. Ma se si chiede di contare quanti locali ci sono in un immobile, alcuni modelli non riescono a distinguere un monolocale da un bilocale. Inoltre, quando i modelli devono dare risposte seguendo un particolare formato, impiegano più tempo a rispondere e a volte le risposte sono qualitativamente inferiori.

Un altro fattore è la dimensione dei documenti da processare in input. Un modello, pur avendo una context length massima sufficientemente grande, con input di grandi dimensioni può produrre risultati peggiori rispetto a un input di piccole dimensioni, perché all'aumentare del contesto aumentano i costi computazionali e peggiorano le capacità di comprensione e ragionamento dei modelli.

Estrazione dati immobiliari

Ad esempio, riguardo all'estrazione dei dati immobiliari dagli avvisi delle aste giudiziarie, i documenti possono essere lunghi decine di pagine. Il file più lungo trovato finora è di quaranta pagine ed equivale a circa 45'000 token. Però i dati relativi agli immobili sono di solito nella prima parte seguiti poi da una sezione in cui viene spiegato come funziona la vendita. Quindi si può rimuovere la seconda parte prima di dare in input il testo al modello. Infatti, si è notato che i file ripuliti della seconda parte vengono elaborati dal modello in meno tempo con risposte più accurate rispetto a quelli grezzi. I test sono stati svolti sfruttando la tecnica del few-shot learning quindi il system e gli esempi aggiungono circa 5000 token, ma questi sono fondamentali affinché qualunque modello possa comprendere il task.

In generale riguardo ai dati specifici, i modelli sono in grado di estrarre dati come indirizzi, prezzi, numeri di procedure giudiziarie e di lotti. Con metri quadrati,

locali, posti auto, vani, piani, dati di identificazione catastale i modelli non sono sempre efficaci. Ad esempio, i modelli fanno spesso fatica a capire quanti locali abbia un immobile, nonostante nel messaggio System si sia data la definizione di locale abitativo con esempi. Nella maggior parte dei casi i modelli riescono a distinguere i

diversi lotti di un avviso giudiziario e ad assegnare i dati a ciascun lotto senza confonderli. I risultati migliori si sono visti all'aumentare dei parametri dei modelli con quantizzazione a INT8 che ha permesso di mantenere un buon livello di accuratezza velocizzando di molto i tempi di risposta. Infatti, Llama3.1:8b-instruct-q8_0 e qwen2.5:7b-instruct-q8_0 hanno avuto prestazioni molto simili a quelle di ChatGPT con GPT-4o. Però non è dato sapere la quantizzazione usata da ChatGPT, né le caratteristiche del sistema su cui gira, nemmeno i parametri del modello che effettivamente risponde alle query.

Il tipo di formato richiesto per l'output influisce sulle prestazioni per due motivi. Primo, formati che richiedono più caratteri di altri, ad esempio XML con i tag di apertura e chiusura, sono più complessi da seguire per i modelli e ci sono più rischi di mal formattazione. In XML i tag non possono contenere spazi e alcuni caratteri, ma i LLM non sempre rispettano ciò, a volte non sono nemmeno in grado di scrivere i tag di chiusura uguali a quelli di apertura. Secondo motivo, i maggiori caratteri richiesti da un formato possono rallentare la velocità di risposta. Una risposta data in JSON è più rapida di una data in XML. Comunque ci sono strumenti che permettono di avere gli output in JSON senza errori di formattazione.

Hugging Face

Hugging Face è una piattaforma per la comunità del machine learning per la collaborazione tra utenti e per la raccolta di modelli, dataset e applicazioni. Il sito fornisce tutorial, risorse e strumenti per machine learning, deep learning e LLM, dalla preparazione dei dati fino al fine-tuning. Fornisce strumenti come huggingface_cli e huggingface_hub da usare in combo con pyTorch o TensorFlow per l'utilizzo dell'IA. Senza avere adeguate conoscenze di machine learning e deep learning è sconsigliabile usare le risorse di sviluppo di Hugging Face, ma resta utile per accedere a modelli e dataset comunque utilizzabili da altre piattaforme

Su https://huggingface.co/leinad-deinor sono stati raccolti dataset e modelli fine-tuned con Unsloth su quegli stessi dataset con l'obiettivo di specializzare i modelli all'estrazione di dati di immobili. Però c'è stato poco miglioramento rispetto ai modelli instruct, fine-tuned per seguire istruzioni, con few-shot learning.

Ollama

Ollama è uno strumento open source per l'utilizzo in inferenza di LLM, semplice da usare e integrare. Ollama essendo basato su llama.cpp è in grado di distribuire il carico dei modelli in inferenza tra GPU e CPU rendendolo ottimo per soluzioni locali in combinazione con modelli open source. Permette di creare dei file, chiamati modelfile, in cui si possono impostare parametri, messaggio System ed esempi, così da poter personalizzare i LLM alle proprie esigenze. Si può definire un JSON schema per l'output strutturato. È pensato per essere usato anche in modalità server con REST API. Di seguito viene fatta una breve introduzione su come usare Ollama.

Si consiglia di approfondire leggendo il README e la guida sul GitHub di Ollama.

CLI

Per usare un modello con Ollama da linea di comando basta usare il comando **ollama run nome-modello**. Prima viene scaricato il modello, se non era già stato scaricato. Poi viene avviata la chat col modello.

È possibile creare dei modelfile dove poter definire il modello da usare, impostando i parametri, scrivendo il System e gli esempi necessari per il prompt engineering. Così quando si chiama il modello modificato dal modelfile con **ollama run**, si ha già tutto impostato nella context length all'avvio senza dover riscrivere ogni volta System ed esempi.

Con il comando **ollama create nome-modello-modificato -f percorso-modelfile** viene creato il modello modificato sulla base del modelfile specificato, che può essere avviato con **ollama run nome-modello-modificato**.

Prompt engineering

In Ollama il prompt engineering è completamente gestibile con i file di tipo modelfile. Un modelfile è una sequenza di uno o più comandi **INSTRUCTION** seguito dagli **argomenti**. L'unica istruzione fondamentale è **FROM** seguita dal nome di un modello, dal percorso dove si trova un modello o dal link di un sito da cui fare il pull del modello, di solito da Hugging Face.

Ad esempio: **FROM llama3.2:1b-instruct-fp16**

I tipi di INSTRUCTION sono:

- **FROM** per definire il modello base da usare. (unica **INSTRUCTION** obbligatoria).
- **PARAMETER** serve per impostare un parametro del modello, ad esempio, la context length massiama, **PARAMETER num_ctx 4096**.
- **TEMPLATE** specifica l'intero template del modello da mandare a Ollama. Di solito si usa quello standard di un modello.
- **ADAPTER** definisce i (Q)LoRA da applicare al modello.
- **LICENSE** specifica la licenza legale.
- **MESSAGE** serve per scrivere la cronologia dei messaggi. Si usa per scrivere gli esempi necessari per il few-shot learning. Quindi si usa almeno una coppia di **MESSAGE**, uno dello user, l'altro dell'assistente.

Consigli per scrivere un modelfile.

È bene partire dal modelfile del modello base e modificarlo. Quindi usare il comando **ollama show nome-modello --modelfile** per ottenere il file. Modificare per primo l'argomento di **FROM** con il nome-modello che si vuole usare. Poi aggiungere o modificare il **SYSTEM**, i **PARAMETER** e i **MESSAGE**.

Nel **SYSTEM** bisogna indicare al modello come deve esprimersi, qual è il suo task e come eseguire il task suddividendolo in sub-task se troppo complesso. Le frasi devono essere semplici e chiare il più possibile. È bene evitare le frasi che contengono negazioni perché potrebbero causare confusione. Nel **SYSTEM** si può già specificare se il modello in output deve rispondere seguendo un determinato formato, come il JSON. Però i modelli non sono molto affidabili in questo ambito, perché a volte dimenticano di generare qualche pezzo che rende il formato valido oppure aggiungono frasi al di fuori del formato. Questo problema viene risolto con altri strumenti.

Un paio di parametri, al di là della context window, a cui è bene prestare attenzione sono **temperature** e **stop**, impostabili con l'istruzione **PARAMETER. Temperature** è un numero reale positivo che indica al modello quanto può essere creativo. Con i task che richiedono rigore, come l'estrazione dati, è consigliabile impostarla a 0, es.

PARAMETER temperature 0. **Stop** è una stringa o espressione regolare, quando il modello la incontra smette di elaborare il prompt. Ciò è utile con file molto lunghi che non si vuole processare per intero con un modello. Es. **PARAMETER stop "Abstract".**

Per fare few-shot learning sono sufficienti da uno a cinque esempi. È importante includere esempi significativi di diversi tipi di interazione che il modello potrebbe avere. Con gli esempi non si può esagerare perché vanno a occupare la context window disponibile insieme al **SYSTEM**.

Server

Ollama può essere usato in modalità server con REST API. La porta di default è 11434, che può essere modificata con le variabili d'ambiente del sistema operativo. Di solito Ollama parte autonomamente in modalità server, ma può essere attivata manualmente con il comando **ollama serve**. Tra i dati della richiesta deve essere specificato almeno il modello, che può essere uno custom creato tramite modelfile con il comando **ollama create**.

Per la richiesta cURL possono essere specificati diversi parametri, due su cui porre l'accento sono **stream** e **format**.

Il parametro **stream** è un booleano per indicare se l'output deve essere rappresentato come uno stream di oggetti che contengono i singoli token della risposta, come si vede fare con le app di chatbot, altrimenti viene prodotto un solo oggetto con l'intera risposta, opzione preferibile quando si sta producendo file.

Il parametro **format** permette di definire un JSON schema per strutturare l'output del modello. I modelli capiscono che devono completare il JSON con i dati presenti nel prompt. Ciò garantisce che la risposta del modello segua il formato senza alcun errore di formattazione. Però, per l'estrazione dati, non garantisce che i dati estratti siano corretti. È comunque consigliabile spiegare al modello il task nel **SYSTEM** del modelfile accompagnandolo con esempi per migliorare il più possibile la qualità della risposta.

Fine-tuning con Unsloth

Il fine-tuning può richiedere molte risorse hardware e tempo. I modelli per essere fine-tuned in genere richiedono quasi il doppio delle risorse hardware necessarie per l'inferenza. Inoltre, è necessario dedicare tempo a curare i dataset necessari per il fine-tuning. Per il primo problema Unsloth riduce l'hardware necessario perchè sfrutta diverse tecniche di ottimizzazione computazionale. Poi, se si è fortunati su Hugging Face forse sono già presenti uno o più dataset adatti all'obiettivo del fine-tuning che si vuole fare. I notebook colab di Unsloth sfruttano la GPU NVIDIA T4 fornita con la versione gratuita di Google Colab, per il quale basta avere un account gmail. Però il tempo di utilizzo della GPU è limitato, quindi non si possono fare fine-tuning che richiedono tanto tempo e tante risorse, ma almeno è utile per fare delle prove.

I notebook vanno semplicemente seguiti passo per passo facendo poche modifiche solo ad alcuni attributi, come i token di accesso e gli indirizzi Hugging Face per salvare i modelli fine-tuned. Poi questi modelli possono essere usati con Ollama mettendo come attributo dell'**INSTRUCTION FROM** nei modelfile il link al modello su Hugging Face. Es. **FROM hf.co/leinad-deinor/Qwen2.5-7B-redeIT-JSON-GGUF.**

Repository GitHub LLM-data-extraction

In questo repository di GitHub è stato raccolto il materiale usato e creato durante il tirocinio. In particolare: articoli scientifici, link utili, modelfile ollama, test, codici Python, script Bash e un applicativo PHP.

Per usare il repository basta usare il comando **git clone https://github.com/LeinadAro/LLM-data-extraction.git .**

Nella cartella **data** sono raccolti i dataset e i file che sono stati usati per la loro creazione. I file sono un'elaborazione degli avvisi giudiziari della cartella **testi_bs**. Per ripulire gli avvisi giudiziari delle parti superflue, che non contengono i dati degli immobili, sono stati usati i codici Python contenuti in **src/python_utils.**

In **python_utils** ci sono solo file per l'elaborazione e la pulizia di file. Per sviluppare un applicativo in Python che sfrutta LLM ci sono molti strumenti come Ollama, pyTorch, TensorFlow, LangChain, e Hugging Face. Però come primo approccio si è

preferito creare dei semplici script Bash contenuti in **bash-scripts**, che sfruttano Ollama. Due lo usano semplicemente da linea di comando, altri due fanno la richiesta al server locale. Per quelli che sfruttano la modalità server è stato impostato l'output strutturato con un JSON schema, pensato per essere usato con un modello creato con un modelfile con le istruzioni per l'estrazione di dati di immobili.

Nella cartella **modelfile** sono raccolti i modelfile basati sui modelli sui quali sono stati fatti la maggior parte dei test con diverse quantizzazioni. Questi modelfile istruiscono i modelli all'estrazione di dati di immobili e sono pronti per essere usati con il comando **ollama create nome-model –f modelfile** senza doverli modificare.

In **src, redeOllama** (real estate data extraction Ollama) è un applicativo PHP che sfrutta Ollama in modalità server. Perciò è necessario che sulla macchina locale sia già stato fatto il run, pull o create di uno o più modelli. L'applicativo ha il file strutturato con un JSON schema per l'estrazione di dati immobiliari. Quindi è consigliabile creare dei modelli usando i modelfile della cartella **modelfile** o simili per poter usare al meglio l'applicativo. L'utente dell'applicativo può selezionare uno dei modelli presenti sul server e poi selezionare uno o più file di testo da far elaborare al modello. Alla fine dell'elaborazione viene scaricato un file zip con le risposte JSON del modello.

Conclusioni

I Large Language Model sono un'importante tecnologia che ha preso piede nel mondo tech, soprattutto per generazione di contenuti, assistenti virtuali, traduzione, analisi dei sentimenti e ricerca di mercato, anche come assistenti di programmazione. Creare applicazioni basate su LLM è abbastanza semplice come si può vedere con redeOllama. Però, per quanto riguarda le soluzioni con modelli open source in locale, bisogna prestare molta attenzione ai requisiti di sistema e trovare un buon rapporto tra costi e benefici. Un modello da 70B di parametri potrebbe avere ottime prestazioni nell'estrazione dati ma richiede una macchina con almeno due GPU da 24 GB di VRAM in parallelo o una GPU con più di 40 GB di VRAM. I modelli lightweight, che funzionano tranquillamente sui PC commercializzati come da gaming, a volte possono dare risultati soddisfacenti altre volte no, come evidente dai test raccolti nel repository GitHub. Le organizzazioni ad ogni nuova versione delle famiglie dei propri modelli apportano importanti modifiche nei metodi di raccolta dei dati e addestramento dei

modelli. Quindi, si spera che le prestazioni dei modelli open source lightweight migliorino fino ad essere paragonabili agli attuali pesi massimi.

Riferimenti

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin, Attention is all you need, https://doi.org/10.48550/arXiv.1706.03762

Humza Naveeda, Asad Ullah Khana, Shi Qiub, Muhammad Saqibc, Saeed Anware, Muhammad Usmane, Naveed Akhtarg, Nick Barnesh, Ajmal Miani, A Comprehensive Overview of Large Language Models, https://doi.org/10.48550/arXiv.2307.06435

L. Chen et al., A Method for Extracting Information from Long Documents that Combines Large Language Models with Natural Language Understanding Techniques, https://ieeexplore.ieee.org/document/10551039

A. Gillioz, J. Casas, E. Mugellini, O. A. Khaled, Overview of the Transformer-based Models for NLP Tasks, https://ieeexplore.ieee.org/document/9222960

Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, Shmargaret Shmitchell, On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?, https://dl.acm.org/doi/10.1145/3442188.3445922

Zhi Rui Tam1, Cheng-Kuang Wu1, Yi-Lin Tsai1, Chieh-Yen Lin1, Hung-yi Lee, Yun-Nung Chen, Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models, https://doi.org/10.48550/arXiv.2408.02442

H. Touvron et al., LLaMA: Open and Efficient Foundation Language Models, https://doi.org/10.48550/arXiv.2302.13971

H. Touvron et al., Llama 2: Open Foundation and Fine-Tuned Chat Models, https://doi.org/10.48550/arXiv.2307.09288

H. Touvron et al., The Llama 3 Herd of Models, https://doi.org/10.48550/arXiv.2407.21783

https://ollama.com/

https://github.com/ollama/ollama

https://huggingface.co/

https://huggingface.co/leinad-deinor

https://github.com/LeinadAro/LLM-data-extraction

https://unsloth.ai/introducing

https://github.com/unslothai/unsloth

https://www.langchain.com/

https://github.com/ggerganov/llama.cpp