UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

YUMI - CONVERSATIONAL AGENT FOR SMART DEVELOPMENT

RELATORE

Prof. **Fabio Palomba**

CO-RELATORE

Dott. **Stefano Lambiase**

Università degli Studi di Salerno

CANDIDATO

Daniele Giaquinto

Matricola: 0512109226

Anno Accademico 2021-2022

"You are never too old to set another goal, or dream a new dream."

C.S. Lewis

Abstract

I recenti progressi nelle tecnologie hardware e di comunicazione, soprattutto lo sviluppo delle architetture su infrastruttura Cloud e del Machine Learning, stanno stravolgendo il modo di interagire tra macchina e uomo.

Lo scopo di questo lavoro di tesi è quello di esplorare queste tecnologie ed apprenderne le capacità al fine di sviluppare "YUMI", un conversational agent per lo smart development che aiuta lo sviluppatore. Tale software, infatti, permetterà allo sviluppatore di accedere in maniera quasi immediata ad una vasta quantità di funzioni.

Negli ultimi anni abbiamo assistito ad un incremento dell'uso dei chatbot in vari campi, ed è proprio a partire dal concetto di "chatbot" che è stata progettata e sviluppata una loro evoluzione, orientata all'aiuto nello sviluppo, che si integra perfettamente anche con altri tool come gli IDE (Integrates Development Environments).

L'ingegnerizzazione del software, ovvero la generazione automatica del codice sorgente, è il sogno di una vita per molti ricercatori dell'ambito informatico e del machine learning.

Oggi questa ambizione sembra diventare sempre più reale grazie a modelli di linguaggio come GPT-3. A partire dall'idea di creazione di un linguaggio di questo tipo, semplificandola, è stato pensato YUMI, un bot creato a supporto degli sviluppatori che permette di suggerire al programmatore una possibile funzione implementabile attraverso l'utilizzo di un'interfaccia.

La piattaforma su cui si basa Yumi è un'applicazione distribuita che interagisce con gli utenti mediante un'interfaccia grafica. Il sistema proposto è basato su uno stile architetturale Client-Server. Tale architettura è perfetta per lo sviluppo dell'applicazione desktop e permette in futuro la facile integrazione all'interno di un IDE poiché separa la logica di business dalla logica di presentazione.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Motivazioni e Obiettivi

I recenti progressi nelle tecnologie hardware e di comunicazione, soprattutto lo sviluppo delle architetture su infrastruttura Cloud e del Machine Learning, stanno stravolgendo il modo di interagire tra macchina e uomo. Questo non solo avviene nell'ambito sociale e quindi a scopo di aiutare l'utente finale, ma anche e soprattutto nell'aiuto allo sviluppatore stesso che crea queste tecnologie.

I campi di ricerca del machine learning come il Deep Learning stanno cambiando in modo marcato e velocemente lo scenario con il quale gli sviluppatori si trovano ad interagire. Se prima gli IDE[1] erano semplici editor di testo corredati con qualche aiuto al programmatore, come la semplificazione del debug del codice, la compilazione e scrittura, oggi, anche grazie al machine learning, stanno diventando sempre più un partner indispensabile per lo sviluppo rapido e per diminuire il time-to-market e i bug nel software.

Lo scopo di questa tesi è quello di esplorare queste tecnologie e apprenderne le capacità con il fine di sviluppare un conversational agent per lo smart development che aiuta lo sviluppatore ad ottenere, tramite l'utilizzo di un modello di intelligenza artificiale e dato un input da parte dello sviluppatore, in output il blocco di codice relativo all'input. Tale software

[1]Integrated Development Enviromen, ovvero un ambiente di sviluppo software che racchiude in un unica interfaccia grafica tutte le funzionalità necessarie alla creazione di codice per svariati linguaggi di programmazione.

permetterà allo sviluppatore di accedere in maniera quasi del tutto immediata ad una vasta quantità di funzioni.

1.2 Risultati

Lo sviluppo del conversational agent Yumi, ha portato alla creazione di un ottimo tool su cui poter lavorare ulteriormente in studi futuri per poter migliorare la sua implementazione. Si è evidenziato come, nonostante il mancato utilizzo dei modelli di linguaggio di dimensione più grande, come per l'appunto il già rinomato GPT-3, gli output di Yumi sono comunque utilizzabili per lo sviluppo di un'estensione per IDE. Uno svantaggio di avere un tool che non utilizza il training per la creazione del modello, ma la cosine similarity per la ricerca intelligente del codice in output, è quello di avere una velocità minore in caso di dataset di grandi dimensioni fino ad arrivare anche all'impossibilità di utilizzarlo in dataset che superano il milione di righe. Questo è derivato dal fatto che il dataset va comunque caricato tutto in memoria sul server che lo utilizza per poterne confrontare la similiarità del commento inserito con il codice trovato. Il vantaggio è per l'appunto quello di poter utilizzare questo tool con dataset più piccoli e direttamente in locale senza dover appoggiare ad architetture che espongono API dietro le quali lavorano anche multipli calcolatori con alte prestazioni. Il tool può essere riadattato a qualsiasi contesto e non obbligatoriamente utilizzato al contesto del program synthesis in quanto anche in uno scenario in cui vi è bisogno della ricerca di similiarità di testo all'interno di documenti, può essere di grande aiuto e facilmente riadattabile grazie agli strumenti di sviluppo utilizzati. Sia il client che il server sono disponibili pubblicamente sulla repository GitHub al link `https://github.com/exSnake/yumi`.

1.3 Struttura della tesi

Il seguente lavoro di tesi è incentrato sullo sviluppo di un Conversational Agent per l'aiuto agli sviluppatori nella generazione di codice.

Nel capitolo 2 verrà mostrato lo stato dell'arte dello sviluppo dei modelli di Machine Learning attualmente in uso. In particolare si parlerà dell'anatomia di un Conversational Agent e l'utilizzo degli stessi nello smart development e di come essi stanno cambiando il modo di interagire con gli ambienti di sviluppo.

Verranno descritti i Dataset che sono al momento presenti, i quali vengono utilizzati per la creazione, il training e il testing di questi modelli, ovvero per valutarne la loro efficacia.

Verranno illustrati i modelli di machine learning già presenti, utilizzati per la creazione di Bot già esistenti come GitHub Copilot[2], Tabnine[3], InCoder[4], ecc. Verrà fatta una panoramica generale su come sono stati generati e quale è il loro ruolo nell'odierno panorama informatico. Nel capitolo 3 è illustrato lo sviluppo di YUMI, un conversational agent che permette di ricevere, dato un input, in output il blocco di codice. Verrà mostrato il processo di creazione del dataset, le scelte architetturali prese e i risultati ottenuti da esse.

[2]Sviluppato da Github, utilizza OpenAI Codex per suggerire intere funzioni in real-time direttamente dall'IDE `https://github.com/features/copilot`

[3]Sviluppato da Tabnine, si integra perfettamente con molti linguaggi nell'aiuto al completamento del testo https://www.tabnine.com/

[4]Descritto nel capitolo 2.4.4

CAPITOLO 2

Stato dell'arte

Questo capitolo illustra lo stato dell'arte e i lavori presenti in letteratura sugli aspetti di ricerca trattati nel nostro studio.

Dagli articoli presenti in letteratura è emerso che l'uso dei chatbot si è evoluto rapidamente in numerosi campi negli ultimi anni, tra cui marketing, sistemi di supporto, istruzione, assistenza sanitaria, beni culturali e intrattenimento. I chatbot sono anche conosciuti come robot intelligenti, agenti interattivi, assistenti digitali o entità di conversazione artificiali. Essi possono imitare la conversazione umana e intrattenere gli utenti, sono anche utili in applicazioni riguardanti istruzione, reperimento di informazioni, affari ed e-commerce. Sono diventati così popolari perché ci sono molto vantaggi sia per utenti che per sviluppatori. [3]

L'interesse per lo sviluppo dei chatbot è in aumento e con esso è cresciuto anche il numero di studi sperimentali condotti sull'usabilità dei chatbot. In generale, i risultati sperimentali hanno rivelato che i chatbot hanno diversi vantaggi, forniscono una risposta in tempo reale e migliorano la facilità d'uso, ma anche diverse carenze, ad esempio, l'elaborazione del linguaggio naturale, è considerata la debolezza più bisognosa di miglioramento.[4]

A partire dal concetto di "chatbot" è stata sviluppata una loro evoluzione, orientata all'aiuto nello sviluppo del software negli Integrates Development Environments (IDEs). Una caratteristica fondamentale all'interno di questi ultimi è l'aiuto che forniscono nel completamento del codice. Sviluppare software di alta qualità riducendo il time-to-market è uno degli obiettivi più ambiziosi all'interno dell'industria di sviluppo software.

Per ridurre il time-to-market è necessario incrementare la produttività degli sviluppatori

riducendo il tempo di apprendimento di un linguaggio e migliorare il codice scritto prevenendo potenziali bug in fase di scrittura. In risposta a queste necessità, i ricercatori hanno proposto dei sistemi per l'ingegnerizzazione del software inseguendo il sogno di una vita, ovvero la generazione automatica del codice sorgente. [5]

Una delle estensioni per IDE più interessanti è Github Copilot per Visual Studio Code, esso fornisce allo sviluppatore dei suggerimenti basati sulla descrizione di un problema da risolvere e sul codice esistente. Basato su un modello per linguaggi in larga scala, Codex, un ulteriore sviluppo di GPT-3 [2], il cui training è stato effettuato su codice sorgente, GitHub Copilot è molto di più che un semplice aiuto nel completamento del codice dato dall'IDE, in quanto non solo può suggerire il completamento del codice, ma anche un un'intera funzione basandosi sulle richieste dello sviluppatore scritte in linguaggio naturale. [6]

```
'''
Function to unizip a file
'''
```

Code Listing 2.1: Commento scritto dallo sviluppatore

```
'''
Function to unizip a file
'''
import zipfile
import os
import sys
import shutil

def unzip(filename):
    zip_ref = zipfile.ZipFile(filename, 'r')
    zip_ref.extractall('.')
    zip_ref.close()
```

Code Listing 2.2: Codice generato da GitHub Copilot

2.1 Anatomia di un Conversational Agent

Un'architettura di riferimento per un conversational agent è quella che vede l'utilizzo dei seguenti componenti:

- 1) **Riconoscimento Vocale**. Se l'applicazione fornisce un'interfaccia vocale con la quale l'utilizzatore può fornire dei comandi, è necessario che il conversational agent sia in grado di trasformare, tramite l'utilizzo di un apposito componente, il parlato in input in un comando da fornire al bot e la risposta in output in una clip audio da fornire al richiedente.[7]

- 2) **Intents e Parametri**. Gli intents sono delle classi di input predefinite che vengono assegnate ai bot da parte degli sviluppatori. ovvero quello che l'applicativo si aspetta di ricevere come comando. I parametri invece, sono degli attributi ricavati dalla lettura degli input e servono ad effettuare un'azione o dare una risposta, qualche esempio nella tabella sottostante:

Input	**Intent**	**Parametri**
Voglio prenotare un albergo	Prenotazione	*Null*
Voglio prenotare un treno da Milano a Salerno il 29/02/2023	Prenotazione	Partenza: Milano Destinazione:Salerno Data: 2023-02-29
Siete aperti a cena?	Richiesta	Dettaglio: Orario
Prodotto impeccabile!	Feedback	Sentimento: Positivo

  La classificazione degli intent è gestita da un componente chiamato *Intent Classifiers*. Invece estrarre i parametri è un lavoro fatto dal componente detto *Parameters Extractor*. Ovviamente parametri e intents potrebbero non includere tutte le possibili combinazioni, se non si riuscisse a classificare l'intent, ne esiste uno di "default".

  La maggior parte delle soluzioni in commercio sono già predisposte tramite un modello *blackbox*, è lo sviluppatore a fornire intent, parametri e frasi di esempio grazie ai quali il modello sarà addestrato.

- 3) **Risponditore**: Come nelle conversazioni tra umani, il risponditore si occupa di generare risposte dagli input ricevuti. Ovviamente nel caso in cui l'input non sia riconoscibile la risposta generata sarà quella in cui verrà chiesto all'utente di riformulare la richiesta. La risposta generata può essere statica o dinamica.

  Una risposta statica può essere un tipico saluto di benvenuto, (es. *"Bentornato Mario"*) mentre quella dinamica dovrà essere generata in base ad input e dettagli esterni (es. *"La consegna verrà effettuata entro le ore 22:00"*).

  Una classe di risposte semi-dinamiche sono le domande che il risponditore fa all'utente a seguito di una richiesta (es. *"A che ora vuoi prenotare il tavolo stasera?"*). Il risponditore può inoltre comunicare con altri componenti del sistema per decidere la risposta da dare all'utente.

- 4) **Controllore del Flusso**: Si occupa di mantenere lo stato della conversazione per decidere qual è la prossima azione da compiere. Per esempio può inviare la richiesta di prenotazione della camera se la domanda precedente era *"Vuoi confermare la prenotazione?"* e la risposta dell'utente è *"Si"* o semplicemente decidere di richiedere all'utente nuovamente delle informazioni sulla prenotazione riportando lo stato corrente a qualche domanda precedente. Per fare questo utilizza i *Context Objects* ognuno dei quali mantiene lo stato ad un determinato istante nel tempo.

- 5) **Azione ed Eseguibili**: il lavoro dei componenti è quello di fornire un interfaccia tramite linguaggio naturale all'utente, per interagire con l'applicazione. Le funzionalità principali che l'applicazione fornisce sono disaccoppiate dai componenti. Queste azioni servono per creare un ponte tra le funzionalità principali dell'applicazione e i componenti. E' il controllore del flusso che chiama un determinato eseguibile basandosi sugli eventi, un evento rappresenta una specifica condizione che accade durante una conversazione con l'utente. Per esempio, se l'intent classifier si accorge che l'utente ha intenzione di prenotare un treno ed il parameter extractor ha estratto tutti i dettagli necessari, il controllore del flusso può eseguire l'azione "l'utente ha richiesto di prenotare il treno".[7]

2.2 Utilizzo di conversational agent nello smart development

Gli sviluppatori che si trovano a risolvere un problema nello sviluppo moderno, raramente si trovano a dover riscrivere il codice da zero.

Piattaforme come Github e StackOverflow, forniscono milioni di esempi di codice per quasi tutte le attività di programmazione immaginabili. L'apprendimento automatico potrebbe rendere il lavoro ancora più semplice.

La sfida per la ricerca è ora trasferire i successi dei modelli linguistici nell'elaborazione del linguaggio naturale ai linguaggi di programmazione con la loro semantica ancor più precisa. Il desiderio di programmi che scrivono programmi è vecchio quanto l' informatica. [8]

Già nel 1957 Alonzo Church, uno dei fondatori della disciplina, all'epoca ancora giovane, formulò la sfida: *"è possibile tradurre automaticamente un'attività da compiere in un programma per computer, cioè tramite un algoritmo, in modo che il programma risolva correttamente l'attività per ogni possibile input?"*.

Oggi a più di 60 anni di distanza la risposta sembra a portata di mano grazie a modelli di linguaggio come GPT-3 (Generative Pre-trained Transformer)[9]

```
def compute_total_price(self, palindrome_discount=0.2):
    """
    Compute the total price.
    Apply a discount to items whose names are palindromes
    """
    total_price = 0
    for item in self.items:
       if is_palindrome(item.name):
          total_price += item.price * (1 - palindrome_discount)
       else:
          total_price += item.price
    return total_price
```

GPT-3 genera codice. Un compilatore da linguaggio naturale, commento in rosso, a codice [10]

2.3 I Dataset

In questo paragrafo è descritto l'utilizzo dei dataset all'interno dei linguaggi di modello, in particolare esamineremo The Pile, HumanEval e APPS questi ultimi due sono Dataset utilizzati per il test dei modelli di program synthesis permettendone di valutarne il grado di risoluzione in problemi di programmazione.

2.3.1 The Pile

The Pile e' un dataset basato su testi in inglese da 825 GiB destinato al training di modelli linguistici su larga scala, esso è stato creato sulla base di 22 dataset preesistenti, che comprendono sia dataset consolidati per l'elaborazione del linguaggio naturale sia alcuni di nuova introduzione.

Il dataset viene creato da varie origini di dati, inclusi i libri; repository GitHub; pagine web; chat; articoli di medicina, fisica, matematica, informatica e filosofia. In particolare, utilizza le seguenti fonti: Pile-CC, PubMed Central, ArXiv, GitHub, FreeLaw Project, Stack Exchange, US Patent and Trademark Office, PubMed, Ubuntu, IRC, HackerNews, YouTube, PhilPapers, Books3, Project Gutenberg (PG-19), OpenSubtitles, Wikipedia in inglese, DM Mathematics, EuroParl, Enron Emails corpus e NIH ExPorter. Include anche OpenWebText2 e BookCorpus2, che sono rispettivamente estensioni dei dataset originali OpenWebText e BookCorpus. La diversità delle fonti di dati può migliorare le conoscenze generali tra domini e di conseguenza migliorare le capacità di generalizzazione a valle.

Oltre alla sua utilità per l'addestramento di modelli linguistici di grandi dimensioni, The Pile può anche servire come benchmark.

Recenti studi hanno dimostrato che, soprattutto per modelli di grandi dimensioni, la diversità delle fonti di dati migliora la conoscenza trasversale del modello e la capacità di generalizzazione. Non solo i modelli addestrati su Pile mostrano miglioramenti nei benchmark tradizionali di modellazione linguistica, ma anche miglioramenti significativi in Pile BPB. Pile BPB (Bits per Bits) è un robusto benchmark della capacità di modellazione testuale generale e trasversale per i modelli linguistici di grandi dimensioni che misura la conoscenza del mondo e della capacità di ragionamento. [11]

La sfida principale con questo dataset è l'enorme dimensione; con i suoi 825 GiB di testo, che si traducono in 4.2 TiB di dati preelaborati e compressi. Si può da subito intuire che il training di un modello con questo dataset su una singola istanza richiederà molto tempo e non è molto pratico.

2.3.2 HumanEval

HumanEval è un dataset di 164 problemi di programmazione scritti a mano, ognuno dei quali include una firma di funzione, documentazione, corpo e alcuni unit test. I problemi di programmazione del dataset valutano la comprensione della lingua, il ragionamento, gli algoritmi e alcuni problemi di matematica semplice. [12] E' stato creato ed utilizzato per testare alcuni modelli di linguaggio adatti al completamento del codice come Codex e GPT-Neo.

2.3.3 APPS

APPS (Automated Programming Progress Standard) è costituito da problemi raccolti da diversi siti web, come Codeforces, Kattis e altri ed è utilizzato principalmente per effettuare benchmark. Il benchmark APPS cerca di rispecchiare il modo in cui vengono valutati i programmatori umani, ponendo problemi di codifica in un linguaggio naturale e utilizzando casi di test per valutare la correttezza della soluzione. I problemi hanno un grado di difficoltà che va da un livello base a quello di competizione Hackathon e misurano la capacità di codifica e di risoluzione dei problemi. (Figura 2.1)

Consiste in un totale di 10.000 problemi di codifica, con 131.777 casi di test per la verifica delle soluzioni e 232.421 soluzioni, scritte a mano. La lunghezza media di un problema è di 293,2 parole, quindi spesso sono anche complicati. I dati sono suddivisi in training-set e test-set, con 5.000 problemi ciascuno. Ogni test-case è progettato in modo specifico per il problema corrispondente, consentendo di valutare rigorosamente la funzionalità del programma.[13]

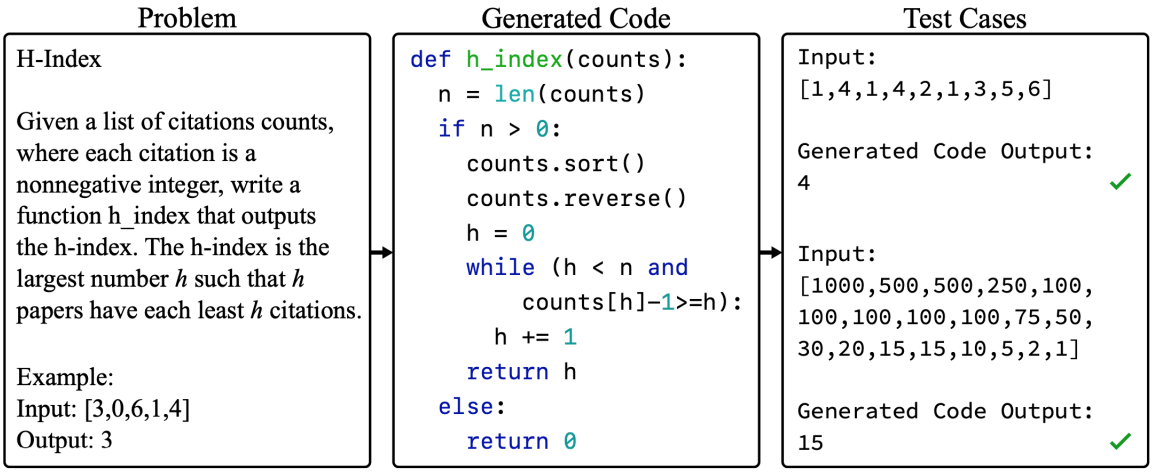

Figura 2.1: Esempio di un problema in APPS (sinistra) insieme ad uno dei possibili codici generati (centro) e due esempi dei test case utilizzati per valutare il codice (destra)

2.4 Modelli di ML per generazione di codice

In questo paragrafo daremo uno sguardo a quelli che sono i modelli NLP per la generazione del codice esistenti al momento della scrittura di questo documento.

2.4.1 GPT-2

GPT-2 è un modello linguistico basato su trasformatore di apprendimento profondo non supervisionato creato da OpenAI nel febbraio 2019 con l'unico scopo di prevedere le parole successive in una frase. GPT-2 è l'acronimo di "Generative Pretrained Transformer 2".

Il modello è open source ed è addestrato su oltre 1,5 miliardi di parametri per generare la successiva sequenza di testo per una data frase. Grazie alla diversità del dataset utilizzato nel processo di addestramento, possiamo ottenere un'adeguata generazione di testo per il testo da una varietà di domini. GPT-2 è 10 volte i parametri e 10 volte i dati del suo predecessore GPT.

Le attività linguistiche come la lettura, il riepilogo e la traduzione possono essere apprese da GPT-2 dal testo grezzo senza utilizzare i dati di addestramento specifici del dominio. Ci sono limitazioni che devono essere considerate quando si ha a che fare con la generazione del linguaggio naturale. I limiti includono testo ripetitivo, incomprensione di argomenti altamente tecnici e specializzati e incomprensione di frasi contestuali.

La lingua e la linguistica sono un dominio complesso e vasto che in genere richiede a un essere umano di sottoporsi ad anni di formazione ed esposizione per comprendere non solo il significato delle parole ma anche come formare frasi e dare risposte che siano contestualmente significative e per usare uno slang appropriato. Questa è anche un'opportunità per creare modelli personalizzati e scalabili per diversi domini. Un esempio fornito da OpenAI è addestrare GPT-2 utilizzando il dataset di Amazon Reviews per insegnare al modello a scrivere recensioni condizionate da cose come la valutazione a stelle e la categoria.

2.4.2 GPT-3

GPT-3 è un modello di linguaggio autoregressivo con 175 miliardi di parametri, 10 volte in più di ogni altro modello di linguaggio sparso. Ottiene risultati grandiosi su parecchi dataset NLP, incluse traduzioni e domande-risposte[2] I linguaggi NLP sono passati da imparare rappresentazioni di problemi e architetture legate in modo specifico ai compiti da eseguire, all'utilizzare training e architetture incuranti del compito da portare a termine. Questo salto ha portato a un progresso sostanziale in molte sfide, come tra le molte, quella di comprendere

quanto letto, rispondere a domande o compiti basati sull'implicazione testuale. Tra i vari tipi di approcci utilizzati da GPT-3 per imparare dal contesto sono stati utilizzati i seguenti:

- **Fine-Tuning (FT)** - Aggiorna i pesi di un modello pre-addestrato effettuando il training su migliaia di label supervisionate specifiche per il compito. Il principale vantaggio del fine-tuning è rappresentato da prestazioni elevate su parecchi benchmark. Gli svantaggi sono la necessità di un nuovo dataset di grandi dimensioni per ogni compito e la scarsa generalizzazione[14] Few-Shot (FS) - Al modello vengono fornite alcune dimostrazioni del compito, ma non vengono aggiornati i pesi. Few-shot funziona dando K esempi di contesto e altrettanti di completamento. Il vantaggio principale è la notevole riduzione della necessità di dati specifici. Lo svantaggio è che i risultati sono peggiori dei modelli con regolazione. fine.[2]
- **One-Shot (1S)** - Simile ad FS ma con $K = 1$.
- **Zero-Shot (0S)** - Simile ad FS ma con una descrizione in linguaggio naturale del compito invece che di esempi. [2]

Setting	LAMBADA (acc)	LAMBADA (ppl)	StoryCloze (acc)	HellaSwag (acc)
GPT-3 Zero-Shot	76.2	3.00	83.2	78.9
GPT-3 One-Shot	72.5	3.35	84.7	78.1
GPT-3 Few-Shot	86.4	1.92	87.7	79.3

Tabella 2.1: Prestazioni di GPT-3 con i vari approcci elencati sopra[2]

Il successo del consolidato strumento di completamento del codice Tabnine dà un'idea del potenziale di questo approccio. [8]

2.4.3 Codex

Codex è un modello di linguaggio creato a partire da GPT sul quale è stato effettuato fine-tuning mediante codice pubblicamente disponibile su GitHub. Il primo tentativo nella creazione di Codex è stato quello di effettuare un fine-tuning sulla famiglia dei modelli di GPT-3, non sono stati però rilevati miglioramenti pertanto successivamente è stato utilizzato GPT come modello di partenza. A differenza di GPT, Codex sul dataset HumanEval ha

mostrato risultati non indifferenti. Infatti riesce a risolvere la maggior parte dei problemi presenti nel Dataset.

	PASS@k		
	$k = 1$	$k = 10$	$k = 100$
GPT-NEO 125M	0.75%	1.88%	2.97%
GPT-NEO 1.3B	4.79%	7.47%	16.30%
GPT-NEO 2.7B	6.41%	11.27%	21.37%
GPT-J 6B	11.62%	15.74%	27.74%
TABNINE	2.58%	4.35%	7.59%
CODEX-12M	2.00%	3.62%	8.58%
CODEX-25M	3.21%	7.1%	12.89%
CODEX-42M	5.06%	8.8%	15.55%
CODEX-85M	8.22%	12.81%	22.4%
CODEX-300M	13.17%	20.37%	36.27%
CODEX-679M	16.22%	25.7%	40.95%
CODEX-2.5B	21.36%	35.42%	59.5%
CODEX-12B	28.81%	46.81%	72.31%

Figura 2.2: Risultati di alcuni modelli su HumanEval

2.4.4 InCoder

InCoder è un modello unificato per la program synthesis. È addestrato a generare file di codice da un ampio corpus di codice, in cui alcune regioni sono state mascherate in modo casuale e spostate alla fine di ogni file. Impara a riempire sostituendo in modo casuale gli intervalli di codice con un token sentinella e spostandoli alla fine della sequenza.(Figura 2.3) Il modello viene addestrato a prevedere tutti i token della sequenza completa. Durante

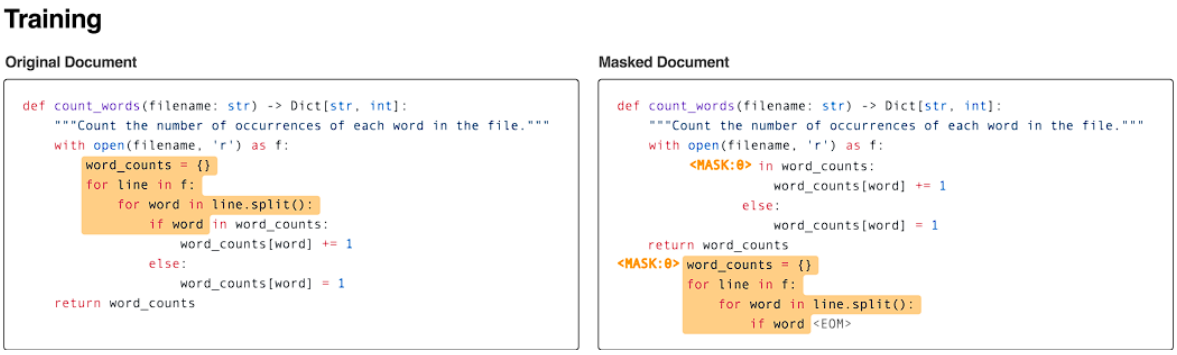

Figura 2.3: Esempio di mascheramento di InCoder

l'inferenza, può modificare il codice sostituendo gli intervalli con i token sentinella, chiedendo al modello la nuova sequenza e facendogli generare nuovi token per sostituire gli intervalli mascherati. (Figura 2.4) I dati su cui è stato effettuato il training di InCoder sono stati presi

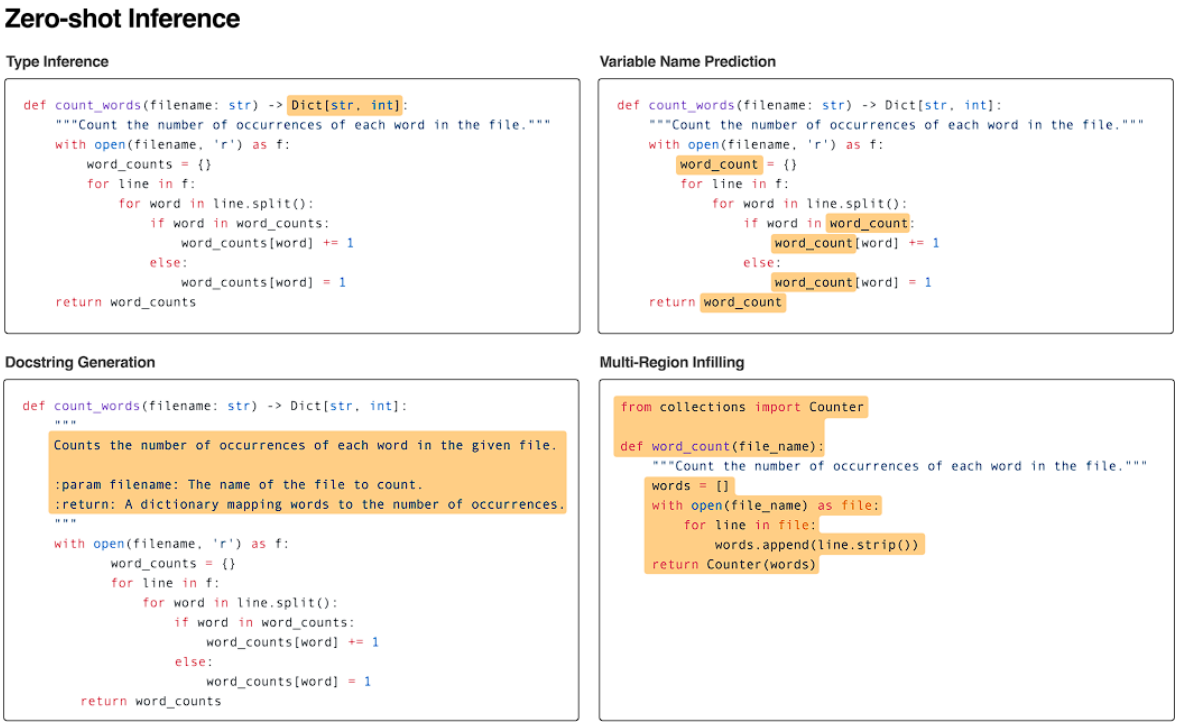

Figura 2.4: Esempio di generazione del codice di InCoder

da codice pubblico open-source da GitHub e GitLab, domande e risposte da StackOverflow con un focus particolare sul linguaggio Python.

2.5 Impatto sociale

Open AI ha descritto accuratamente quale potrebbe essere l'impatto di un modello di linguaggio addestrato su codice esistente presente all'interno dei dataset, come quelli di cui abbiamo parlato nel paragrafo precedente.

- **Eccessiva Fiducia nel Modello**: un modello linguistico addestrato su grandi insiemi di dati con il compito di autogenerare codice può generare soluzioni plausibili che possono sembrare corrette, ma non necessariamente lo sono. La mancata valutazione del codice generato può avere conseguenze negative, come l'introduzione di bug o di vulnerabilità di sicurezza. Pertanto, è importante che gli utenti siano consapevoli dei limiti e delle potenziali conseguenze negative dell'utilizzo di un modello linguistico addestrato su questo set di dati. [12]

- **Impatto economico e sul mercato del lavoro**: i grandi modelli linguistici addestrati su grandi insiemi di codice come GPT-3, in grado di generare codice di alta qualità, hanno il potenziale di automatizzare parte del processo di sviluppo del software. Ciò potrebbe

avere un impatto negativo sugli sviluppatori di software. Tuttavia, come discusso nel documento, e come mostrato nel Rapporto di sintesi sugli sviluppatori di software di O*NET OnLine, gli sviluppatori non scrivono solo software.[12]

- **Implicazioni per la sicurezza**: non è stato effettuato alcun filtro o controllo delle vulnerabilità o del codice difettoso. Ciò significa che il dataset può contenere codice che può essere dannoso o contenere vulnerabilità. Pertanto, qualsiasi modello addestrato su questo set di dati può generare codice vulnerabile, con presenza di bug o dannoso. Questo potrebbe portare a un software che potrebbe funzionare in modo improprio e causare gravi conseguenze a seconda del software. Inoltre, un modello addestrato su questo set di dati potrebbe essere utilizzato per generare di proposito codice dannoso al fine di eseguire ransomware o altri attacchi simili.[12]

- **Implicazioni legali**: non è stato eseguito alcun filtraggio sul codice con licenza. Ciò significa che il dataset può contenere codice con licenza restrittiva. Come discusso nel documento, i repository Github pubblici possono rientrare nel "fair use". Tuttavia, in passato sono stati registrati pochi casi di utilizzo di codice disponibile su licenza. Pertanto, qualsiasi modello addestrato su questo set di dati potrebbe essere tenuto a rispettare i termini di licenza in linea con il software su cui è stato addestrato, come ad esempio la GPL-3.0, motivo per cui abbiamo volutamente messo questo set di dati sotto la licenza GPL-3.0. Non sono chiare le conseguenze legali dell'uso di un modello linguistico addestrato su questo dataset.[12]

2.6 Cosine Similarity

La cosine similarity è una metrica ampiamente implementata nell'information retrieval e negli studi correlati. Questa metrica modella un documento di testo come un vettore di termini. In base a questo modello, la somiglianza tra due documenti può essere ricavata calcolando il coseno tra i vettori di termini dei due documenti.

L'implementazione di questa metrica può essere applicata a due documenti che siano essi frasi, paragrafi o documenti interi. I valori di similarità tra la query e i documenti sono ordinati dal più alto al più basso, una similarità elevata tra il vettore del documento e quello della query indica una maggiore pertinenza tra i due.

La cosine similarity per valutare la somiglianza tra il documento e la query dell'utente dovrebbe adattarsi al significato della parola; tuttavia, ancora non si è in grado di gestire perfettamente il significato semantico del testo.

L'implementazione della cosine similarity, a volte, produce risultati inaffidabili dal punto di vista sintattico. Per risolvere questo problema sono stati condotti studi sulla misurazione semantica o sulla somiglianza semantica tra le parole. Il metodo più comune utilizza un database lessicale come rete semantica cioè WordNet.

Attraverso l'esplorazione di questo database si può ricavare la somiglianza tra due concetti, basandosi sul giudizio umano e non solo sulla cosine similarity. WordNet si propone di modellare la conoscenza lessicale di una madrelingua inglese. L'unità più piccola di WordNet è un gruppo logico chiamato synset, che rappresenta il senso di una parola. I synset hanno relazioni semantiche tra loro. Un documento può essere descritto in forma vettoriale come:

$$\vec{d} = (W_{d0}, W_{d1}, ..., W_{dk})$$

La query può essere descritta in forma vettoriale:

$$\vec{q} = (W_{q0}, W_{q1}, ..., W_{qk})$$

Dove W_{di} e W_{qi} $(0 \leq i \leq k)$ sono float che indicano la frequenza di ogni termine all'interno di un documento, mentre la dimensione di ogni vettore corrisponde a un termine disponibile nel documento. In base alla cosine similarity, la similarità tra due vettori può essere definita come:

$$Sim(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}|\left|\vec{d}\right|} = \frac{\sum_{k=1}^{t} W_{qk} \times W_{dk}}{\sqrt{\sum_{k=1}^{t} \left(W_{qk}\right)^2} \cdot \sqrt{\sum_{k=1}^{t} (W_{dk})^2}}$$

La Cosine similarity può essere applicata solo a vettori della stessa dimensione altrimenti otterremmo un valore di similarità bassa, quando il valore di similarità è troppo basso è diverso dal giudizio umano. Attualmente sono in corso studi al fine di migliorare la similarità del coseno tra due vettori di termini basandosi sul giudizio umano.

Nello scenario sopra descritto, la somiglianza del coseno tendente a `1` implicherebbe che i due documenti sono esattamente simili mentre una somiglianza del coseno tendente a `0` porterebbe alla conclusione che non ci sono somiglianze tra i due documenti. Vediamo un esempio:

```
Frase 1: Il deep learning può essere difficile
Frase 2: Il deep learning può essere semplice
```

Per prima cosa otteniamo una rappresentazione vettorializzata dei testi, per farlo, si analizzano le singole parole distinte con significato all'interno della frase, eliminando da esse congiunzioni e articoli:

Deep	Learning	può	essere	difficile	semplice

Dopodiché per ogni frase si va ad assegnare alla posizione `i-esima` del vettore un valore `1` oppure `0` rispettivamente se la parola è presente all'interno della frase o meno. Chiamiamo i due documenti A e B, otteniamo quindi la seguente rappresentazione vettoriale:

	Deep	Learning	Può	Essere	Difficile	Semplice
A	1	1	1	1	1	0
B	1	1	1	1	0	1

Utilizzando questi vettori e la formula vista in precedenza possiamo procedere con il calcolo della consine similarity.

$$Sim(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\left|\vec{A}\right| \left|\vec{B}\right|}$$

Andiamo dapprima a calcolare il prodotto scalare tra $\vec{A}$ e $\vec{B}$:

$$\vec{A} \cdot \vec{B} = 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 = 4 \tag{2.6.1}$$

Successivamente calcoliamo la norma dei due vettori:

$$\left|\vec{A}\right| = \sqrt{1^2 + 1^2 + 1^2 + 1^2 + 0^2} = 2.2360679775 \tag{2.6.2}$$

$$\left|\vec{B}\right| = \sqrt{1^2 + 1^2 + 1^2 + 0^2 + 1^2} = 2.2360679775 \tag{2.6.3}$$

Pertanto la cosine similarity sarà uguale a:

$$Sim(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\left|\vec{A}\right| \left|\vec{B}\right|} = \frac{4}{2.2360679775 * 2.2360679775} \simeq 0.8 \tag{2.6.4}$$

Ovvero da questo possiamo dedurre che abbiamo una somiglianza pari all'80% tra le frasi estrapolate dai due documenti.

CAPITOLO 3

Design

3.1 Obiettivi

Questo capitolo mostra le scelte di design e lo sviluppo di Yumi. Lo scopo è quello di creare un bot a supporto degli sviluppatori che permetta di suggerire al programmatore una possibile funzione implementabile attraverso l'utilizzo di un'interfaccia che accetti un input e restituisca la funzione in output. Grandi modelli basati su linguaggio naturale hanno dimostrato grandi progressi nella neural program synthesis e task correlati. Le scelte di creazione del nostro program synthesis andavano in due direzioni diverse. L'utilizzo di modelli pre-addestrati e delle loro Api con l'aggiunta del nostro dataset accorpato a quello già presente, oppure un approccio diverso basato sulla similiarità del testo in input con quello ricercato. Tuttavia la prima soluzione avrebbe portato a diverse difficoltà:

- l'impossibilità di utilizzo di alcuni framework che avrebbero reso possibile questa scelta
- l'impossibilità di utilizzo del nostro calcolatore per il training in quanto la potenza di calcolo necessaria non era sufficente
- l'impossibilità di trovare la mole di dati necessaria ad utilizzare il fine-tuning

Pertanto, si è deciso di affrontare il problema in esame mediante tecniche di Machine Learning che permettessero di effettuare un calcolo basato sulla similarità dei vari elementi.

3.2 Funzionalità e processo di sviluppo

In questo paragrafo si esaminerà il modello di sviluppo e i casi d'uso di Yumi, nello specifico i due casi d'uso GetConcreteConversation e GetRandomConversation e il modello di sviluppo Test-Driven Development utilizzato per la scrittura del codice. Il sistema proposto è un software scritto in Flutter che ha lo scopo di aiutare lo sviluppatore nella ricerca di una determinata funzione basandosi su un testo da lui scritto.

3.2.1 Casi d'uso

In questo paragrafo verranno mostrati i casi d'uso per Yumi. I casi d'uso (use cases) sono descrizioni di come può essere usato un sistema, servono a raccogliere i requisiti del software in maniera esaustiva e non ambigua, focalizzandosi sugli attori che interagiscono col sistema, valutandone le varie interazioni.

L'attore principale in questo caso è proprio l'utente. Esso avrà la possibilità di effettuare delle query al modello del nostro conversational agent e ricevere da lui delle risposte. Un'altra funzionalità è quella di ricevere delle risposte randomiche, quest'ultima pensata perlopiù per valutarne le potenzialità e l'effettiva funzionalità.

Il secondo attore per l'appunto è il sistema, le cui funzionalità, derivate dai requisiti funzionali, vengono descritte tramite i casi d'uso.

In Yumi abbiamo due casi d'uso:

- **GetConcreteConversation**: Che servirà per ottenere, dato un input, in output da parte del modello il relativo blocco di codice più consono a quanto richiesto dall'utente. Questo è il caso d'uso principale del nostro conversational agent, in quanto sfrutta tutte le funzionalità per il quale è stato creato, ovvero la predizione del blocco di codice dall'input dell'utente. (Tabella 3.1)

- **GetRandomConversation**: Che servirà per ottenere, in output da parte del modello un blocco di codice ricavato in modo randomico da parte del modello all'interno del nostro dataset. Questo caso d'uso è stato creato per scopi scientifici, sarà utilizzato principalmente per testare il corretto funzionamento del conversational agent e delle API ad esso legate e valutarne le potenzialità.(Tabella 3.2)

Tabella 3.1: Tabella Caso d'uso GetConcreteConversation

Use Case #1	**GetConcreteConversation**		
Descrizione	Ottenere il codice di una funzione		
Entry Condition	L'utente deve trovarsi nella schermata principale		
Exit on Success	La piattaforma restituisce in output un blocco di codice		
Exit on Failure	La piattaforma restituisce un messaggio di errore		
Attore Primario	Utente		
Trigger	Inserimento dell'input e pressione del comando		
FLUSSO DI EVENTI	**Step**	**Attore 1**	**Sistema**
	1	L'utente usa il comando per scrivere un commento	
	2		Il sistema
FLUSSO ALTERNATIVO	**Step**	**Attore 1**	**Sistema**
	1	L'utente usa il comando per scrivere un commento	
	2		Il sistema mostra un messaggio di errore
NOTE			

Tabella 3.2: Tabella Caso d'uso GetRandomConversation

Use Case #1	**GetRandomConversation**		
Descrizione	Ottenere il codice casuale di una funzione		
Entry Condition	L'utente deve trovarsi nella schermata principale		
Exit on Success	La piattaforma restituisce in output un blocco di codice casuale		
Exit on Failure	La piattaforma restituisce un messaggio di errore		
Attore Primario	Utente		
Trigger	Pressione del comando per l'output casuale		
FLUSSO DI EVENTI	**Step**	**Attore 1**	**Sistema**
	1	L'utente usa il comando per ottenere un commento casuale	
	2		Il sistema mostra il blocco di codice relativo al commento
FLUSSO ALTERNATIVO	**Step**	**Attore 1**	**Sistema**
	1	L'utente usa il comando per ottenere un commento casuale	
	2		Il sistema mostra mostra un messaggio di errore
NOTE			

3.2.2 Modello di Sviluppo

Si è scelto di seguire un modello di sviluppo Test-Driven Development (abbreviato TDD) che prevede che la creazione di test avvenga prima della scrittura del software che deve essere sottoposto a test. Più in dettaglio il TDD prevede la ripetizione di un breve ciclo di sviluppo in tre fasi, detto "Ciclo TDD" (Figura 3.1).

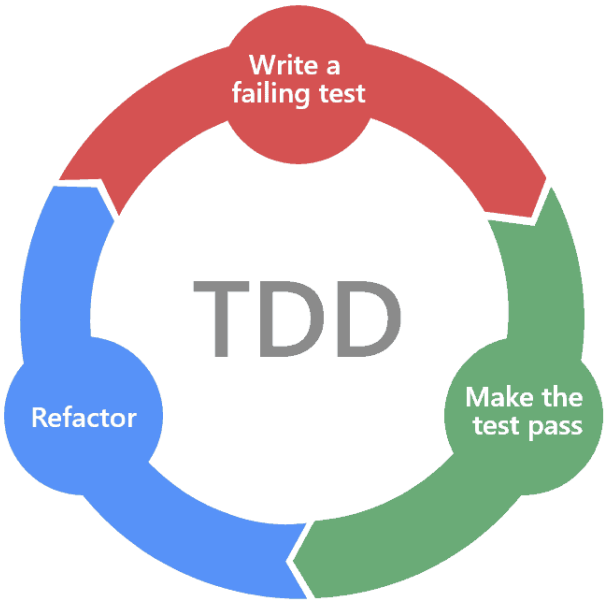

Figura 3.1: Le tre fasi del Test-Driven Development

- **Red Phase**: In questa fase viene scritto un test, nel nostro caso in flutter utilizzando il package `flutter_test/flutter_test.dart`, volto a validare la funzionalità da implementare. Una volta completato il test che descrive la funzionalità che verra implementata, deve essere eseguito e all'esecuzione dovrà produrre un errore (si dice che il test è Rosso/Red). In questo momento la fase è completata e si passa alla successiva del ciclo.

- **Green Phase**: In questa fase si implementa la funzionalità con la minima quantità di codice possibile volto a far si che il test passi (si dice che il test è Verde/Green). Il codice non deve essere di qualità, e deve focalizzarsi sullo sviluppo della parte necessaria a far passare il test, infatti è vietato nello sviluppo TDD aggiungere codice che non è finalizzato al superamento del test. In questa fase inoltre si eseguono anche tutti i test precedenti per far si che la funzionalità appena creata non abbia intaccato parti di funzionalità precedentemente scritte. Quando tutto è Verde (Green) si passa alla fase successiva.

- **Refactoring Phase**: Quando tutti i test sono Verdi si passa al refactoring del codice, ovvero a migliorare la sua leggibilità e struttura attraverso un procedimento basato su piccole modifiche, per esempio, l'eliminazione del codice duplicato, la semplificazione di cicli, rinominare variabili e funzioni etc. L'obiettivo non è quello di ottenere codice perfetto ma solo di migliorarne la struttura.

```dart
import 'dart:convert';

import 'package:flutter_test/flutter_test.dart';
import 'package:yumi/features/conversation/data/models/conversation_model.dart';
import 'package:yumi/features/conversation/domain/entities/conversation.dart';

import '../../../../fixtures/fixture_reader.dart';
void main() {
    const tConversationModel = ConversationModel(number: 1, text: 'Test Text');
    group('fromJson', () {
        test(
          'should return a valid model wheen the JSON number is an integer',
          () async {
            // arrange
            final Map<String, dynamic> jsonMap =
                json.decode(fixture('conversation.json'));
            // act
            final result = ConversationModel.fromJson(jsonMap);
            // assert
            expect(result, tConversationModel);
          },
        );
      });
    }
}
```

Code Listing 3.1: Un esempio di test all'interno del progetto per la creazione del modello

L'applicazione del TDD porta in generale allo sviluppo di un numero maggiore di test e a una maggiore ed è stato possibile applicarlo al progetto in quanto è abbastanza semplice da fare in questo caso, trattandosi di un software minimale. Infatti, se da un lato il TDD permette di scrivere un'applicazione quasi esente di bug, dall'altro il tempo impiegato per portare a termine la creazione di una funzionalità aumenta, soprattutto in team piccoli.

3.3 Il Dataset

Il dataset necessario al modello di intelligenza artificiale per suggerire un possibile output in base al commento in input è stato generato selezionando i repository GitHub da un'ampia raccolta di repository. Queste repository sono state raccolte da SEART-GithubSearch `https://seart-ghs.si.usi.ch/`[1]. L'obiettivo di questo dataset è fornire un set di addestramento per il preaddestramento di modelli linguistici di grandi dimensioni su dati di codice per aiutare i ricercatori di ingegneria del software a comprendere meglio il loro impatto sulle attività relative al software, come il completamento automatico del codice. Contiene principalmente codice Dart, ma sono inclusi anche altri linguaggi di programmazione in varia misura. È stato il lavoro più importante ed anche il più delicato in quanto ha richiesto numerose ore tra la ricerca dei tool adatti all'esportazione dei dati, l'esportazione dei dati stessi che come vedremmo superano gli 80Gb, fino alla creazione degli script di ingegnerizzazione degli stessi.

3.3.1 Struttura del Dataset

In questa sezione andiamo ad esaminare la struttura del dataset risultante.

```
{
  "id": datasets.Value("int64"),
  "comment": datasets.Value("string"),
  "commentSize": datasets.Value("int64"),
  "code": datasets.Value("string"),
  "codeSize": datasets.Value("int64"),
  "repo": datasets.Value("string"),
}
```

I campi del dataset mostrati nel formato json di cui sopra sono identificati in:

- **id**: Identificatore unico per il dato.
- **comment**: Il testo del commento associato alla funzione.
- **commentSize**: La lunghezza in caratteri, spazi inclusi, del commento.
- **code**: Il testo della funzione descritta dal commento.
- **codeSize**: La lunghezza in caratteri, spazi inclusi, del codice.
- **repo**: Il nome della repository GitHub dalla quale è stato estratto.

3.3.2 Creazione del Dataset

Il primo passo è stato quello di estrapolare da SEART-GithubSearch il CSV contenente i dati delle repository che combaciavano con la ricerca effettuata utilizzando i seguenti criteri:

- >10 GitHub stars
- >5 commits
- Must have license
- Exclude Forks
- Language: Dart

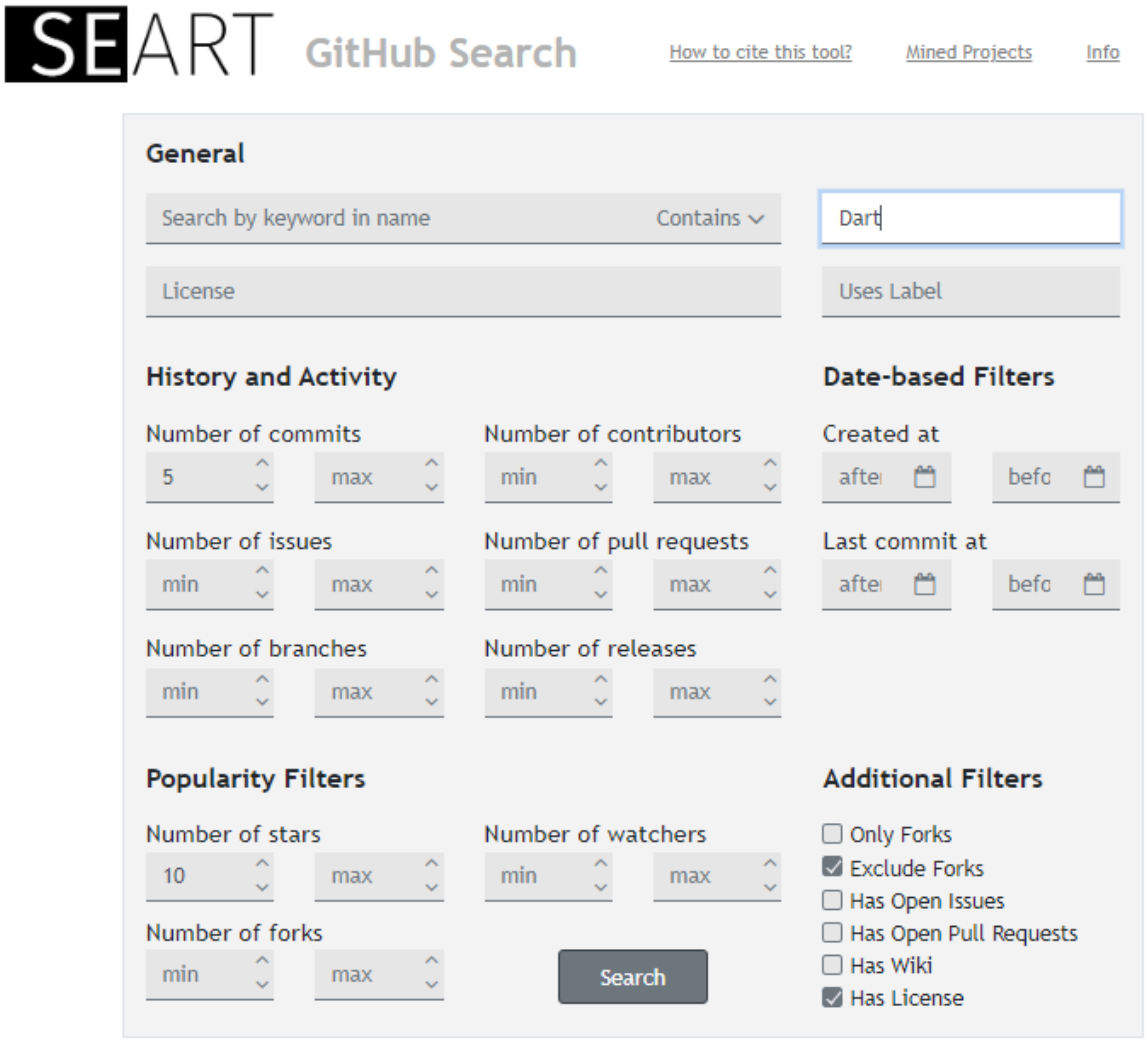

Figura 3.2: Interfaccia Web di SEART-GithubSearch `https://seart-ghs.si.usi.ch/`[1].

Il file estrapolato è suddiviso in questo modo: `name, isFork, commits, branches, defaultBranch, releases, contributors, license ...` da questo è stato creato un secondo file CSV `github_dart_repo.csv` necessario all'estrazione che conterrà solamente l'attributo `name` relativo a `autore\nomeProgetto` della repository GitHub.

Successivamente si è passati alla clonazione di tutte le repository utilizzando uno script in Python di cui riportiamo una parte del codice:

```
absolute_path = os.path.dirname(os.path.abspath(__file__))
def download_repo(repo):
    file_name = repo.split("/")[-1]
    if file_name not in os.listdir("output/"):
        os.system(f'git clone --depth 1 --single-branch https://github.com/{repo}
                                                    output/{file_name}')
    else:
        print(f"Already downloaded {repo}")
with open('github_dart_repo.csv', 'r') as f:
    csv_reader = csv.reader(f)
    repositories = list(map(tuple, csv_reader))
if 'output' not in os.listdir():
    os.makedirs('output')
repo_names = [repo[0] for repo in repositories]
Parallel(n_jobs=40, prefer="threads")(
    delayed(download_repo)(name) for name in tqdm(repo_names))
```

Code Listing 3.2: Codice python per il clone automatico delle repository[1]

Questo script ha generato in una cartella di output tutti i file presenti nelle repository dai quali sono stati poi filtrati i file dart e dai quali sono stati esportati tutti i commenti.

```
ext = ('.dart')
def extractdata():
    i=0
    for root, dirs, files in tqdm(os.walk(directory)):
        for filename in files:
            if filename.endswith(ext):
                path = os.path.join(root, filename)
                parseFileComments(path)
                if len(arrayOfText) > 0:
                    saveComments(repo)
```

Code Listing 3.3: Un'estratto del codice per l'estrazione dei commenti dai file `.dart`

Per dare un'idea del massivo lavoro presente nella costruzione del dataset, nella Figura 3.3 è mostrato il totale dei dati grezzi su cui è stato effettuato il lavoro per estrarre i commenti e il codice relativo:

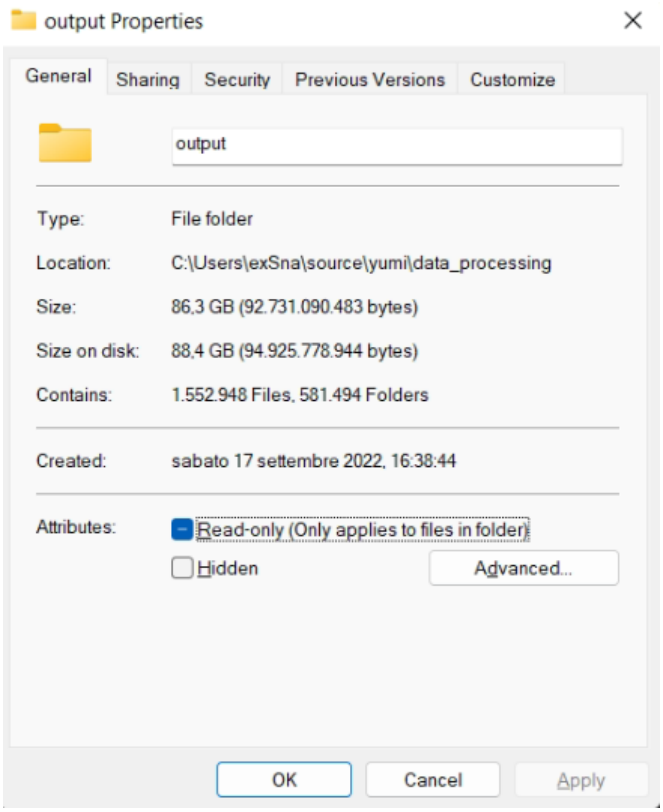

Figura 3.3: Proprietà della cartella Output generato da explorer di Windows

3.3.3 Ingegnerizzazione dei dati

Nella prima fase di ingegnerizzazione dei dati, essi sono stati analizzati per valutarne la validità andando ad eliminare file non relativi a codice. Sono stati elaborati mediante uno script in Python presente nella repository sotto "datacleaning.py" che è stato utilizzato per l'estrazione dei campi elencati precedentemente, da ogni singolo file con estensione .dart, lo script ha poi generato un file di tipo .csv in cui ogni riga del file è associata ad una riga del dataset. Nella seconda fase sono stati analizzati ed eliminati file duplicati mediante uno script in python presente nella repository "deduplication.py", lo script puo essere utilizzato da riga di comando:

```
deduplication.py -data_dir <data_dir> -output_dir <output_dir>
```

Il tool assume che due file sono duplicati se hanno la stessa sequenza di variabili.

I file in input presenti nella directory devono essere in formato `jsonl`. Il tool caricherà ogni file, trasformerà il codice ottenuto in token, ovvero una lista di variabili all'interno del codice.

Le variabili sono ottenute mediante regex di token che non sono di tipo alfanumerico, esse vengono poi utilizzate per scovare duplicati negli altri file. In output ci saranno solo i file che non contengono le stesse variabili.

Un ulteriore cleaning del dataset é stato effettuato eliminando codice pre-generato dagli IDE, come per esempio il codice di generazione di un nuovo progetto. Ottenendo quello che è poi il dataset finale sul quale andremo a lavorare. Nella tabella 3.3 è presente un esempio, generato dal comando `df.sample(n = 10).sort_index()`, delle righe del dataset.

Va fatta una precisazione, i dati sono stati collezionati da repository pubbliche, esiste il rischio che siano presenti informazioni sensibili che gli sviluppatori hanno lasciato all'interno del codice in modo involontario quali secrets key, password, API keys, email, etc. Questo non e' stato oggetto di controllo in quanto, dopo una breve ricerca, non é stato trovato alcun modulo in grado di effettuare un controllo di questo tipo in breve tempo sulla mole di dati a disposizione.

comment	size	code	size	repo
Video mirror mode.\n\n	29	@JsonEnum(alwaysCreate: true)\n	31	agoraio/agora-flutter-sdk
Persists in-memory changes to a file on di...	51	class JsonFileService extends Service<String, ...	437	angel-dart/angel
Minimum contrast ratio	150	const minimumTextContrast = 4.5;\n	34	angulardart/angular_components
Minimum values breakpoints for [DeviceScre...	147	const defaultMinimumBreakPoints = ScreenBreakp...	69	jamescardona11/argo
Class to expose a stream of [Set<AtKey>] b...	986	class SetKeyStream<T>extends KeyStreamIterabl...	64	atsign-foundation/at_client_sdk
Parses english relative time that's in for...	538	DateTime parseEnglishRelativeTime(\n	36	edwardez/bangumin
Fibonacci function ...	127	int fibonacci(int n) => n <= 2 ? 1 :...	388	letsar/binder
A person's or business card...	70	class ContactInfo {\n Gets contact person...	1422	yanshouwang/camerax
class for InterSegmentPointer	64	class InterSegmentPointer extends Pointer {\n ...	600	jonaswanke/capnproto-dart
SQL_AUTOCOMMIT options\n	27	const int SQL_AUTOCOMMIT_OFF = 0;\n	39	jcmellado/dart-odbc

Tabella 3.3: Un esempio dei dati contenuti nel dataset

3.4 L'architettura

La piattaforma su cui si basa Yumi è un'applicazione distribuita che interagisce con gli utenti mediante un'interfaccia grafica. Il sistema proposto è basato su uno stile architetturale Client-Server. Il motivo della presente scelta è che tale architettura è perfetta per lo sviluppo sia dell'applicazione desktop che andremmo ad utilizzare sia di una possibile implementazione futura come web application, poiché separa la logica di business dalla logica di presentazione, migliorando:

- Manutenzione
- Leggibilità
- Riuso
- Portabilità
- Scalabilità

3.4.1 Bloc State Management

Per ottenere ciò utilizzeremo Bloc, uno state management per Flutter che permette di separare la nostra applicazione in tre livelli:

- Presentation Layer
- Domain Layer
- Data Layer
  - Repository
  - Data Provider

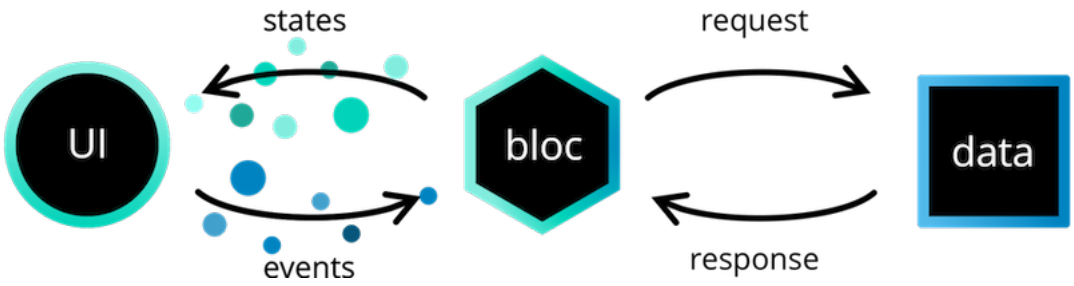

Figura 3.4: Schema dell'architettura

3.4.2 Clean Architecture Principle

Separare la logica in livelli indipendenti è un principio essenziale di progettazione del software, Robert C. Martin anche detto Uncle Bob[15] e il progetto è strutturato proprio seguendo questi principi. (Figura 3.5) Applicare questo principio a Flutter non è proprio

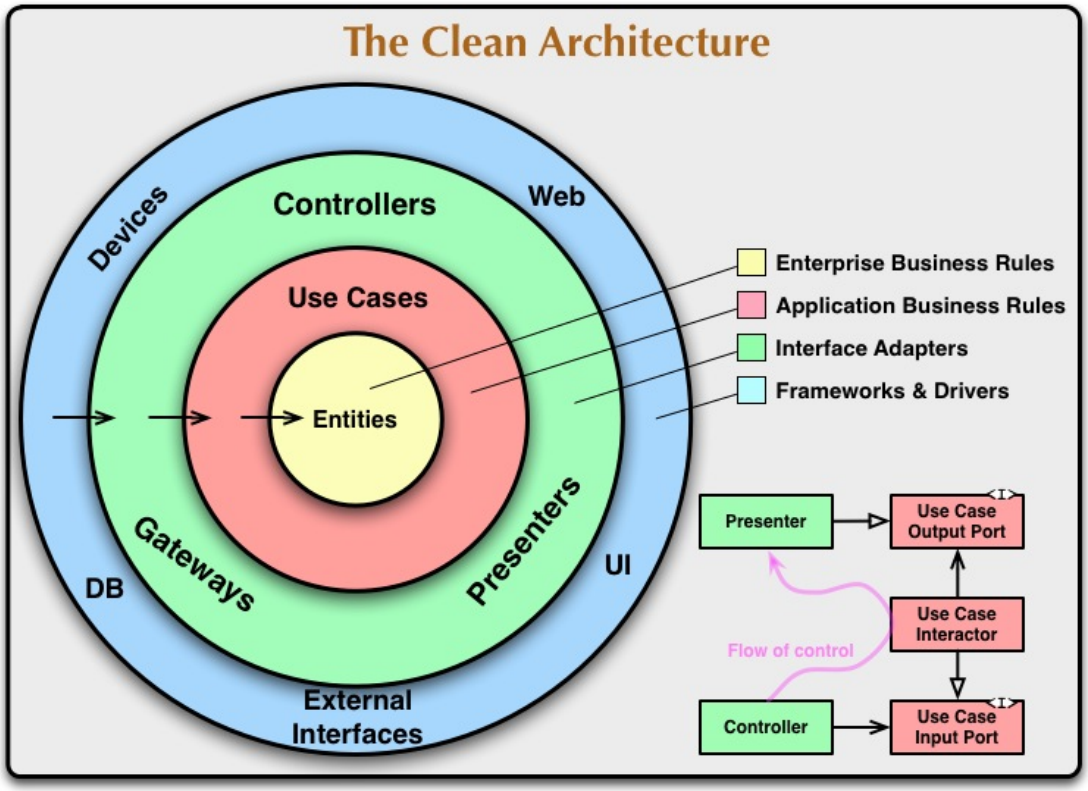

Figura 3.5: Clean Architecture Principle

come presentato nell'immagine presente sul libro, in quanto non sono propriamente presenti Controller o Gateway, ma l'essenza dell'architettura "pulita" resta lo stesso in qualsiasi framework e quello che faremo è applicarlo a Flutter. Per farlo è stato creato un flusso che i dati dividendo l'applicazioni in tre parti fondamentali, il livello di presentazione, il livello di dominio e il livello dei dati (Figura 3.6)

3.4.3 Presentation Layer

La struttura proposta è anche rispecchiata nei namespace creati all'interno del progetto, nel livello di presentazione possiamo trovare i Widgets, ovvero i componenti principali di flutter per l'interfaccia grafica e Bloc, lo state management scelto per lo sviluppo dell'applicazione. Il livello di presentazione non farà molto da solo come è solito fare all'interno dei

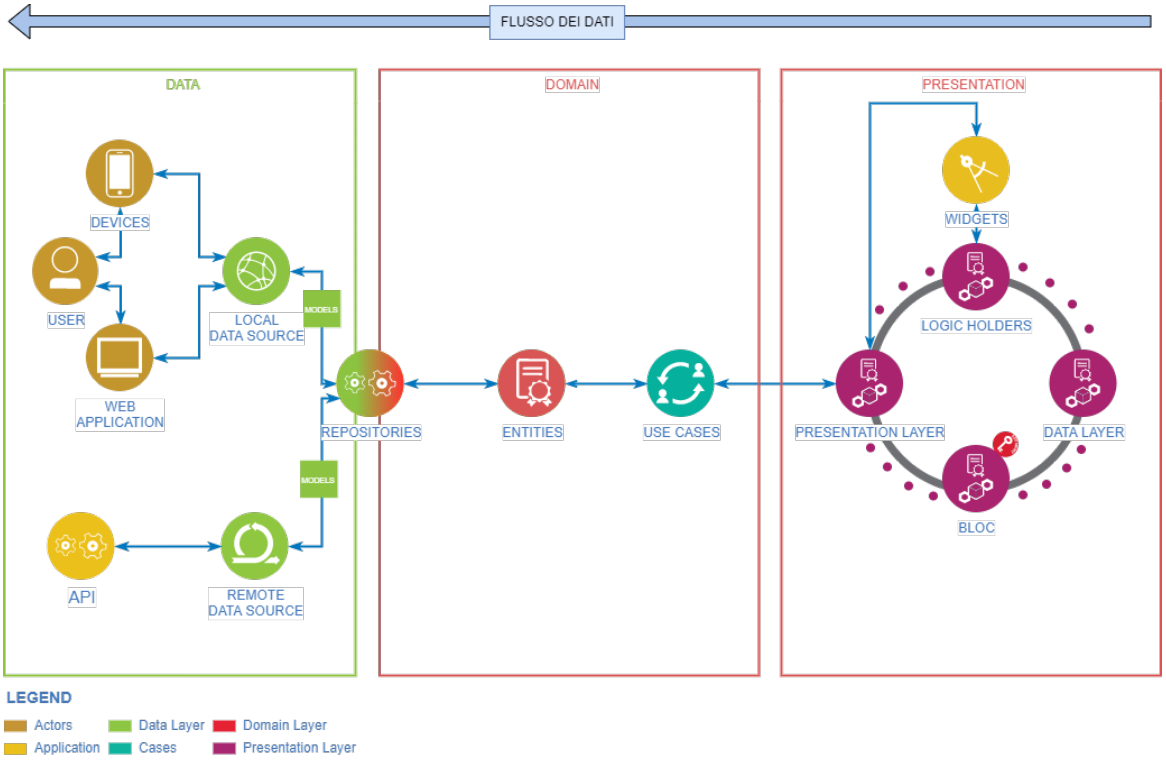

Figura 3.6: I livelli dell'architettura proposta

progetti veloci di Flutter, ma effettueranno al massimo qualche controllo sugli input. Tutto il resto verrà delegato al secondo livello ovvero il Domain Layer attraverso gli eventi generati da Bloc (Code Listing 3.4).

```
class ConversationBloc extends Bloc<ConversationEvent, ConversationState> {
  final GetConcreteConversation getConcreteConversation;
  final GetRandomConversation getRandomConversation;
  final InputConverter inputConverter;
  ConversationBloc({required GetConcreteConversation concrete,
      required GetRandomConversation random,
      required this.inputConverter})
      : getConcreteConversation = concrete,
        getRandomConversation = random,
        super(Empty()) {
    on<GetConversationForConcreteStringEvent>(
        (event, emit) async => _onConcreteStringEvent(event, emit));
    on<GetConversationForRandomStringEvent>(
        (event, emit) async => _onRandomStringEvent(event, emit));
  }
}
```

Code Listing 3.4: Presentation Layer: conversation_bloc.dart

3.4.4 Domain Layer

Bloc delegherà il lavoro al livello di dominio, il livello di dominio non dovrà essere suscettibile al cambio del livello dei dati, da dove essi provengono, ma sarà completamente indipendente da esso, conterrà solamente la logica di business che verrà eseguita all'interno dei casi d'uso e i business object, ovvero le entità. Gli use cases non sono altro che classi che conterranno tutta la logica di business relativa al particolare caso d'uso.

Dalla figura 3.6 possiamo notare come il domain layer contiene sul bordo le repositories che saranno replicate sia nel domain layer che nel data layer e per continuare a mantenere il grado di indipendenza quello che faremo è utilizzare quello che viene chiamato Dependency Inversion.

La Dependency Inversion (Code Listing 3.5) verrà implementata creando delle classi astratte che definiranno un "contratto" su cosa le repository dovranno fare. Questo, nell'ottica del modello di sviluppo Test-Driven Development ci permette di "mockarle" facilmente senza doversi curare della vera implementazione.

Queste classi verranno poi registrate dal `ServiceLocator` che si occuperà di effettuare la dependency injection nei vari componenti.

`import 'package:get_it/get_it.dart'` fornisce la classe `GetIt` che permette di registrare Factory e Singleton con i metodi `registerFactory`, `registerSingleton` oppure `registerLazySingleton` e si può recuperare l'istanza istanziata con il metodo `.get()` oppure direttamente con la chiamata a funzione dell'istanza, avendo internamente implementato la funzione `call()`.

```dart
final sl = GetIt.instance;
Future<void> init() async {
  // Features - Conversation
  //Bloc
  sl.registerFactory(() =>
      ConversationBloc(concrete: sl(), random: sl(), inputConverter: sl()));
  // Use cases
  sl.registerLazySingleton(() => GetConcreteConversation(sl()));
  sl.registerLazySingleton(() => GetRandomConversation(sl()));
  // Repository
  sl.registerLazySingleton<ConversationRepository>(() =>
      ConversationRepositoryImpl(localDataSource: sl(), remoteDataSource: sl(),
    networkInfo: sl()));
  //Data sources
  sl.registerLazySingleton<ConversationRemoteDataSource>(
      () => ConversationRemoteDataSourceImpl(client: sl()));
  sl.registerLazySingleton<ConversationLocalDataSource>(
      () => ConversationLocalDataSourceImpl(sharePreferences: sl()));
}
```

Code Listing 3.5: injection_container.dart

```dart
import 'package:dartz/dartz.dart';

import '../../../../core/error/failure.dart';
import '../../domain/entities/conversation.dart';
import '../../domain/repositories/conversation_repository.dart';

class ConversationImpl implements ConversationRepository {
  @override
  Future<Either<Failure, Conversation>> getConcreteConversation(int number) {
    // TODO: implement getConcreteConversation
    return null;
  }

  @override
  Future<Either<Failure, Conversation>> getRandomConversation() {
    // TODO: implement getRandomConversation
    return null;
  }
}
```

Code Listing 3.6: Data Layer: conversation_repository_impl.dart

3.4.5 Data Layer

Il livello dei dati definirà come i dati vengono ottenuti e gestiti, la repository nel livello dei dati implementerà il contratto definito in precedenza (Code Listing 3.7) e quindi si passerà dalla classe astratta alla vera e propria implementazione (Code Listing 3.6) conformandosi al contratto.

In questo modo il livello di dominio non dovrà preoccuparsi su cosa succede dietro le quinte, ma saprà solamente che tipo di dati riceverà che nel nostro caso sarà il testo del blocco di codice ottenuto in modo diretto o randomico in base ai casi d'uso che più in la sono definiti.

```
1 import 'package:dartz/dartz.dart';
2
3 import '../../../../core/error/failures.dart';
4 import '../entities/conversation.dart';
5
6 abstract class ConversationRepository {
7   Future<Either<Failure, Conversation>> getConcreteConversation(int number);
8   Future<Either<Failure, Conversation>> getRandomConversation();
9 }
```

Code Listing 3.7: Domain Layer:conversation_repository.dart

Nel nostro caso nel livello dei dati c'è sia un Remote Data Source che otterrà i dati dalle API del nostro modello, sia dal Local Data Sources che otterrà i dati in input da parte dell'utente mediante il dispositivo utilizzato, che sia esso un'applicazione Web o Desktop o Mobile. Repository sarà il cervello del Data Layer in quanto deciderà da quale data source e quando prendere i dati. Dalla figura 3.6 possiamo notare come la repository crea l'entità, che nel nostro caso è la risposta dell'API con il blocco di codice in formato testo. I data sources hanno invece in output i models che si occuperanno di gestire oggetti di tipo Json.(Code Listing 3.9)

La logica di conversione da json a entità non verrà messa nell'entità ma nei models proprio per garantire il livello di indipendenza dal livello dei dati (Code Listing 3.8). Se per esempio il data source desse in output anziché degli oggetti json, un markdown xml, dovremmo poter sostituire il livello dei dati senza preoccuparci di modificare le entità e quindi il livello di dominio.

```dart
import '../../domain/entities/conversation.dart';

class ConversationModel extends Conversation {
  const ConversationModel({required text, required number})
      : super(text: text, number: number);
  factory ConversationModel.fromJson(Map<String, dynamic> json) {
    return ConversationModel(text: json['text'], number: json['number']);
  }

  Map<String, dynamic> toJson() {
    return {'text': text, 'number': number};
  }
}
```

Code Listing 3.8: Data Layer:conversation_model.dart

```dart
class ConversationRemoteDataSourceImpl implements ConversationRemoteDataSource {
  final http.Client client;
  final url = 'http://localhost:5001/';
  final headers = {'Content-Type': 'application/json'};

  ConversationRemoteDataSourceImpl({required this.client});

  Future<ConversationModel> _getConversationFromUrl(String url) async {
    final response = await client.get(Uri.parse(url), headers: headers);
    if (response.statusCode == 200) {
      return ConversationModel.fromJson(json.decode(response.body));
    } else {
      throw ServerException();
    }
  }

  @override
  Future<ConversationModel> getConcreteConversation(String comment) =>
      _getConversationFromUrl('http://localhost:5001/query=?$comment');

  @override
  Future<ConversationModel> getRandomConversation() =>
      _getConversationFromUrl('http://localhost:5001/random');
}
```

Code Listing 3.9: Data Layer: comversation_remote_data_source_impl.dart

3.4.6 Frontend

Lato frontend, per ottenere un MVP (Minimum Viable Product) è stata costruita una semplice interfaccia grafica che permetterà, tramite l'utilizzo di due comandi, `Cerca Funzione` e `Funzione Casuale` di ottenere rispettivamente, una funzione ricercata in base al testo inserito dall'utente nell'input-box, richiamando il caso d'uso `getConcreteConversation` ed una funzione casuale ricavata in modo randomico dal dataset, questa racchiude per l'appunto il secondo caso d'uso `getRandomeConversation`.

In entrambi i casi verrà presentato in output un contenitore base (con scrolling nel caso in cui la funzione sia lunga) con all'interno il codice della funzione ed un ulteriore tasto per copiarne il contenuto ed eventualmente permettere allo sviluppatore di utilizzarlo.

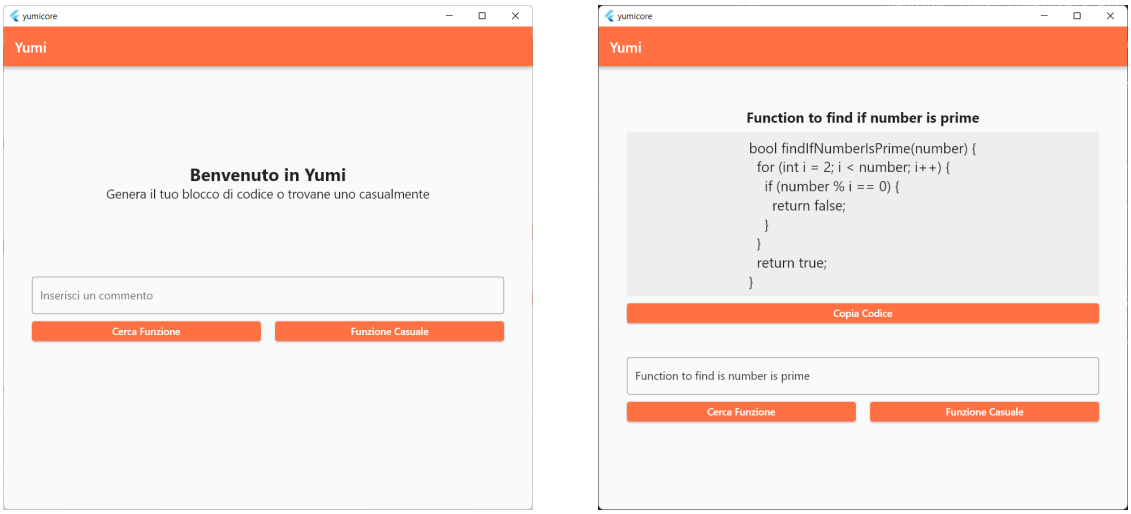

Figura 3.7: Schermata principale di Yumi (sinistra) e codice generato dopo la richiesta (destra)

3.4.7 Backend

Lato backend invece, è stato creato un server che tramite Python espone delle API contenenti due endpoint `random` e `concrete`. La prima ci permetterà di accedere ad un risultato casuale legato al caso d'uso `getRandomConversation`, mentre l'ultima restituirà il risultato della query in input ovvero del caso d'uso sopra descritto `getConcreteConversation`.

Per fare ciò utilizzeremo un server tramite Api RESTFul Flask che risponderà agli URL locali:

```
GET: http://localhost:5001/random

GET: http://localhost:5001/query?comment=testoquery
```

```
class ConcreteConversation(Resource):
    @use_kwargs({'comment': fields.Str(required=True)},location="query")
    def get(self, comment):
        return {
            "comment": "{}!".format(comment),
            "code": "{}".format(getConcreteConversation(comment))
        }

class RandomConversation(Resource):
    def get(self):
        return getRandomConversation()

# Error handler for Flask-RESTful
@parser.error_handler
def handle_request_parsing_error(err, req, schema, *, error_status_code,
                                 error_headers):
    """webargs error handler that uses Flask-RESTful's abort function to return
    a JSON error response to the client.
    """
    abort(error_status_code, errors=err.messages)


if __name__ == "__main__":
    api.add_resource(ConcreteConversation, "/query")
    api.add_resource(RandomConversation, "/random")
    app.run(port=5001, debug=True)
```

Code Listing 3.10: Un'estratto del codice per l'avvio del server backend in locale

Il backend, testato in locale, interagisce con il modello e dal dataset va estrarre in risposta, in formato `json` commento e codice della funzione trovata.

```
{
  "comment": "Find if number is prime",
  "code": "  bool findIfNumberIsPrime(int number) {[...]}"
}
```

3.5 Estendibilità e Sviluppi Futuri

L'utilizzo di quanto visto nel capitolo 3.4 e le tecniche di clean architecture descritte nel paragrafo 3.4.2 e successivi, danno modo non solo al nostro tool di essere esteso in modo veloce e controllato ma anche di essere riutilizzato per altri scopi.

L'interfaccia grafica è completamente separata dalla logica di business, i parametri all'interfaccia arrivano tramite lo state manager adottato, nel nostro caso Bloc, ma anch'esso a sua volta è completamente separato dal livello di dominio, quindi è possibile sostituirlo con qualsiasi altro event manager.

La logica di dominio può essere a sua volta sostituita, andando semplicemente a modificare l'implementazione della repository. Per esempio per cambiare l'API alla quale Yumi fa le richiesta per l'output è immediato, si crea una nuova implementazione della `ConversationRepository` e la si inietta tramite Dependency Injection all'interno del Service Locator, in poche righe di codice Yumi può diventare un generatore di ricette, oppure un vero e proprio chatbot! E così via anche per il livello dei dati e i modelli, possiamo elaborare XML piuttosto che file JSON in qualche minuto.

> *"Non ci vuole poi questa gran conoscenza o abilità per far funzionare un programma. Far funzionare qualcosa, almeno una volta, non è poi così difficile. Far le cose per bene è proprio tutta un'altra cosa. Scrivere un buon software è difficile. Quando il software è fatto bene, la sua creazione e manutenzione richiede solo una frazione delle risorse umane. Ogni modifica è semplice e rapida"*[15]

Non solo, è anche possibile esporre le API direttamente dalla repository piuttosto che interagire con un'interfaccia desktop è possibile far si che Yumi diventi una Web App in pochissime righe di codice.

Gli sviluppatori spesso credono che il time-to-market sia più importante di una buona architettura, la corsa al rilascio del software è molto presente all'interno dell'informatica ma è anche molto dannosa. Il *"possiamo sempre sistemarlo dopo"* fa tendere la produttività a zero, è importante per questo motivo cercare di uscire sin da subito con idee di struttura del codice ben manutenibile e estensibile e Yumi, anche se è un progetto piccolo sviluppato in singolo, ne è un esempio.

CAPITOLO 4

Conclusioni

Questo studio si è posto l'obiettivo di creare un conversational agent a partire da quanto già presente in letteratura e semplificandone la gestione del modello.

Si è focalizzato sul linguaggio DART di cui poco è presente in rete al momento della scrittura. Il dataset stesso, che è stato il lavoro più lungo e articolato, può essere riutilizzato in molti ambiti.

I risultati ottenuti sono piuttosto soddisfacenti e dal tool creato, grazie anche all'adozione di tecniche di clean architecture e del modello di sviluppo TDD si può procedere, sia a migliorare l'interazione con lo sviluppatore, magari integrandolo in un IDE, sia a migliorare le funzionalità del modello, per esempio effettuando il fine-tuning con il dataset generato verso un modello di linguaggio più avanzato come il GPT-3 o i suoi derivati.

In conclusione le strade da poter intraprendere a partire da questo studio sono molte. Possiamo però affermare che il machine learning e l'intelligenza artificiale hanno imboccato una buona direzione attraverso anche l'utilizzo del Deep Learning che sicuramente porterà in futuri piuttosto brevi a risultati di grande rilevanza nella generazione del linguaggio, sia umano (chatbot, linguaggio naturale), sia sintattico (program synthesis, linguaggio di sviluppo).

Anche se le sfumature del linguaggio sono ancora troppe per riuscire ad ottenere un alto livello di dettaglio, il rapido avanzamento nei nuovi modelli sta rendendo molto probabile che la prossima grande svolta possa essere dietro l'angolo, con lo sviluppo delle nuove tecnologie sarà possibile raggiungere tale scopo in un futuro non molto lontano.

La quarta rivoluzione industriale è iniziata e l'intelligenza artificiale farà da pioniere quindi è importantissimo incentivare e sponsorizzare la ricerca e gli studi fatti in questo ambito.

Ringraziamenti

Questo spazio lo dedico alle persone che, con il loro supporto, mi hanno aiutato, supportato e alcuni "sopportato", in questo percorso, sia nell'ambito universitario che nella vita privata.

Ringrazio il mio relatore nonché professore **Fabio Palomba** che mi ha trasmesso le sue conoscenze sia durante il suo corso di Fondamenti di Intelligenza Artificiale, sia durante la stesura di questa tesi, è stato un vero mentore.

Ringrazio il dottore **Stefano Lambiase** che mi ha dato utilissimi suggerimenti per condurre al meglio questo progetto.

I miei compagni universitari. In particolare **Leonardo**, **Rebecca**, **Davide**, **Simone** e il resto del team di FitDiary **Daniele**, **Antonio**, **Ilaria** con cui ho condiviso, chi più, chi meno, questo percorso.

I miei amici, la mia ottima scusa per giustificare il ritardo nel conseguimento di questa Laurea, **Vito**, **Francesco**, **Roberto**, **Daniele**, **Peppe** e **Ivan**, se avessimo realizzato un decimo delle idee pensate durante le serate a girovagare forse oggi non sarebbe stato necessario scrivere queste righe.

Marika, la persona grazie alla quale invece le posso scrivere queste righe, mi ha spronato per anni, ha sempre creduto in me e mi ha aiutato ad arrivare a questo traguardo.

Letizia, **Gerardo** e **Rossella** per aver custodito lo champagne per tutto questo tempo, fiduciosi che un giorno l'avrei aperto e **Nicolino** per non averlo distrutto con la bicicletta.

Per finire mia **Mamma**, mio **Padre** e i miei fratelli **Antonio**, **Pasquale**, **Gianluca** per l'appoggio che mi han dato durante tutta la vita e tutti i miei nipoti, **Salvatore**, **Francesco**, **Mattia**, **Christian**, **Tommaso** e **Lorenzo**.

Bibliografia

[1] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564. (Citato alle pagine iv, 24 e 25)

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020. (Citato alle pagine v, 5, 11 e 12)

[3] E. Adamopoulou and L. Moussiades, "An overview of chatbot technology," in *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, 2020, pp. 373–383. (Citato a pagina 4)

[4] R. Ren, M. Zapata, J. W. Castro, O. Dieste, and S. T. Acuña, "Experimentation for chatbot usability evaluation: A secondary study," *IEEE Access*, vol. 10, pp. 12 430–12 464, 2022. (Citato a pagina 4)

[5] F. Wen, E. Aghajani, C. Nagy, M. Lanza, and G. Bavota, "Siri, write the next method," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 138–149. (Citato a pagina 5)

[6] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming," *arXiv preprint arXiv:2111.07875*, 2021. (Citato a pagina 5)

[7] S. Srivastava and T. Prabhakar, "A reference architecture for applications with conversational components," in *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2019, pp. 1–5. (Citato alle pagine 6 e 7)

[8] B. Finkbeiner and F. Schmitt, "Künstliche intelligenz in der softwareentwicklung: Über die schulter geschaut," *iX Magazin für professionelle Informationstechnik*, no. 8, pp. 40–43, 2021. (Citato alle pagine 8 e 12)

[9] S. Chintala. (2020, May) Openai showed an cool code generation demo at msbuild2020 of a big language model trained on lots of github repositories the demo does some non-trivial codegen specific to the context.eagerly waiting for more details!video: Starting at 28:45 https://t.co/k1oapwh6hc. [Online]. Available: https://twitter.com/soumithchintala/status/1263221177650159620 (Citato a pagina 8)

[10] F. Crivello. (2020, Jul) Gpt3 writing code. a compiler from natural language to code.people don't understand - this will change absolutely everything. we're decoupling human horsepower from code production. the intellectual equivalent of the discovery of the engine. https://t.co/qgjbqrbdqv pic.twitter.com/cjiark8j0m. [Online]. Available: https://twitter.com/Altimor/status/1278736953836400640 (Citato a pagina 8)

[11] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The Pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020. (Citato a pagina 9)

[12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374 (Citato alle pagine 10, 14 e 15)

[13] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with APPS,"

CoRR, vol. abs/2105.09938, 2021. [Online]. Available: https://arxiv.org/abs/2105.09938 (Citato a pagina 10)

[14] R. T. McCoy, E. Pavlick, and T. Linzen, "Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference," *arXiv preprint arXiv:1902.01007*, 2019. [Online]. Available: https://arxiv.org/abs/1902.01007 (Citato a pagina 12)

[15] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st ed. USA: Prentice Hall Press, 2017. (Citato alle pagine 30 e 38)