UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

CADOCS 2.0: Evoluzione e trasformazione di CADOCS in una desktop application

RELATORE

Prof. Fabio Palomba

Dott. Stefano Lambiase

Dott. Gianmario Voria

Università degli Studi di Salerno

CANDIDATO

Paolo Carmine Valletta

Matricola: 0512112622

Anno Accademico 2022-2023

Questa tesi è stata realizzata nel

"Ripenso a prima, mi guardo ora
Da quando stavo fuori scuola ne ho fatta di strada"

Abstract

I *community smells* sono caratteristiche socio-tecniche in una comunità di sviluppo, che possono finire per ostacolare la produzione complessiva del software, che se reiterati nel tempo, portano a social debt, quindi a compromettere la produttività e l'efficienza di un team di svilupppo.

È importante rilevare i community smells perchè una rilevazione efficiente e tempestiva degli stessi può svolgere un ruolo cruciale nell'aiutare i project manager a preservare la salute della comunità di sviluppo. Lo scopo principale di questo progetto di tesi è stato effettuare un lavoro di manutenzione su uno degli strumenti più utilizzati dagli sviluppatori per rilevare i community smells, ovvero csDetector. In parallelo, è stato condotto un lavoro di evoluzione su CADOCS, un chatbot che effettua il wrap di csDetector per fornire funzionalità aggiuntive.

Nel corso di questo lavoro sono state apportate diverse modifiche a csDetector per ottimizzare la gestione degli errori, migliorando la chiarezza delle informazioni in caso di malfunzionamenti. Inoltre è stato sviluppato un installer per rendere il processo di installazione del tool più intuitivo e accessibile.

Per quanto riguarda la seconda parte del nostro lavoro, CADOCS è stato trasformato in una desktop app, tramite l'utilizzo della libreria Tkinter per l'interfaccia grafica e Pyinstaller per la creazione del file eseguibile, integrando al proprio interno il modulo NLU per la predizione degli intent e un sistema di login tramite GitHub.

Complessivamente, le modifiche apportate hanno contribuito a ottimizzare l'identificazione e la gestione dei community smells, fornendo strumenti più efficaci per migliorare la qualità e l'efficienza dello sviluppo software.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

I *community smells* sono caratteristiche socio-tecniche in una comunità di sviluppo, che se reiterati nel tempo, portano a social debt. Possono comprendere aspetti sociali come mancanza di comunicazione, discrepanze nelle competenze del team, o lacune nella documentazione che possono finire per ostacolare la produzione complessiva del software, compresa la manutenzione e l'evoluzione. Il problema è individuare questi smells al meglio, la rilevazione accurata dei Community smells può fornire un quadro dettagliato delle aree problematiche all'interno del processo di sviluppo del software. Affrontare queste criticità può contribuire significativamente alla riduzione del debito sociale e tecnico, migliorando la sostenibilità a lungo termine e la qualità complessiva del software.

Individuare e interpretare i Community smells all'interno di una comunità di sviluppo software costituisce un processo complesso. Questi aspetti socio-tecnici, pur essendo fondamentali per comprendere le dinamiche di un team di sviluppo, spesso sfuggono all'analisi. Tuttavia, esiste già un sistema in grado di rilevare queste criticità, tramite un insieme di metriche: csDetector.

1.2 Problema: Community smells portano a social debt

I *social debt* sono tutti quei costi e i problemi derivanti da una cattiva gestione delle dinamiche sociali e delle relazioni all'interno di una comunità di sviluppo o di un team di programmatori. Questo debito può verificarsi quando, per esempio, ci sono conflitti non risolti o mancanza di comunicazione efficace.

È interessante notare che dietro al concetto di social debt, si riscontrano sempre i Community smells. Questi rappresentano una delle principali cause che contribuiscono alla formazione e all'accumulo del debito sociale nell'ambito della comunità di sviluppo software. È importante quindi rilevare, venire a conoscenza di questi community smells per cercare di risolverli.

1.3 CADOCS e sue limitazioni

Dall'identificazione del problema precedentemente specificato, si è evoluto lo sviluppo di un chatbot integrato alla piattaforma Slack, progettato per rilevare i Community smells. Questo sistema, noto come CADOCS, funge da 'interfaccia' per csDetector, arricchendolo con ulteriori funzionalità proprie di un chatbot tradizionale.

Tuttavia, csDetector presenta diverse criticità, come una gestione degli errori inefficace: in caso di problemi durante l'esecuzione, il tool tende a interrompersi senza fornire dettagli o spiegazioni sui malfunzionamenti. Allo stesso modo, la sua installazione risulta complessa e problematica.

Di conseguenza, queste problematiche hanno un impatto diretto su CADOCS, poiché il chatbot si basa su di esso. In aggiunta alle criticità condivise con csDetector, CADOCS presenta ulteriori problematiche, come quelle relative all'installazione e all'integrazione all'interno della piattaforma di chat Slack.

1.4 Obiettivo e lavoro svolto

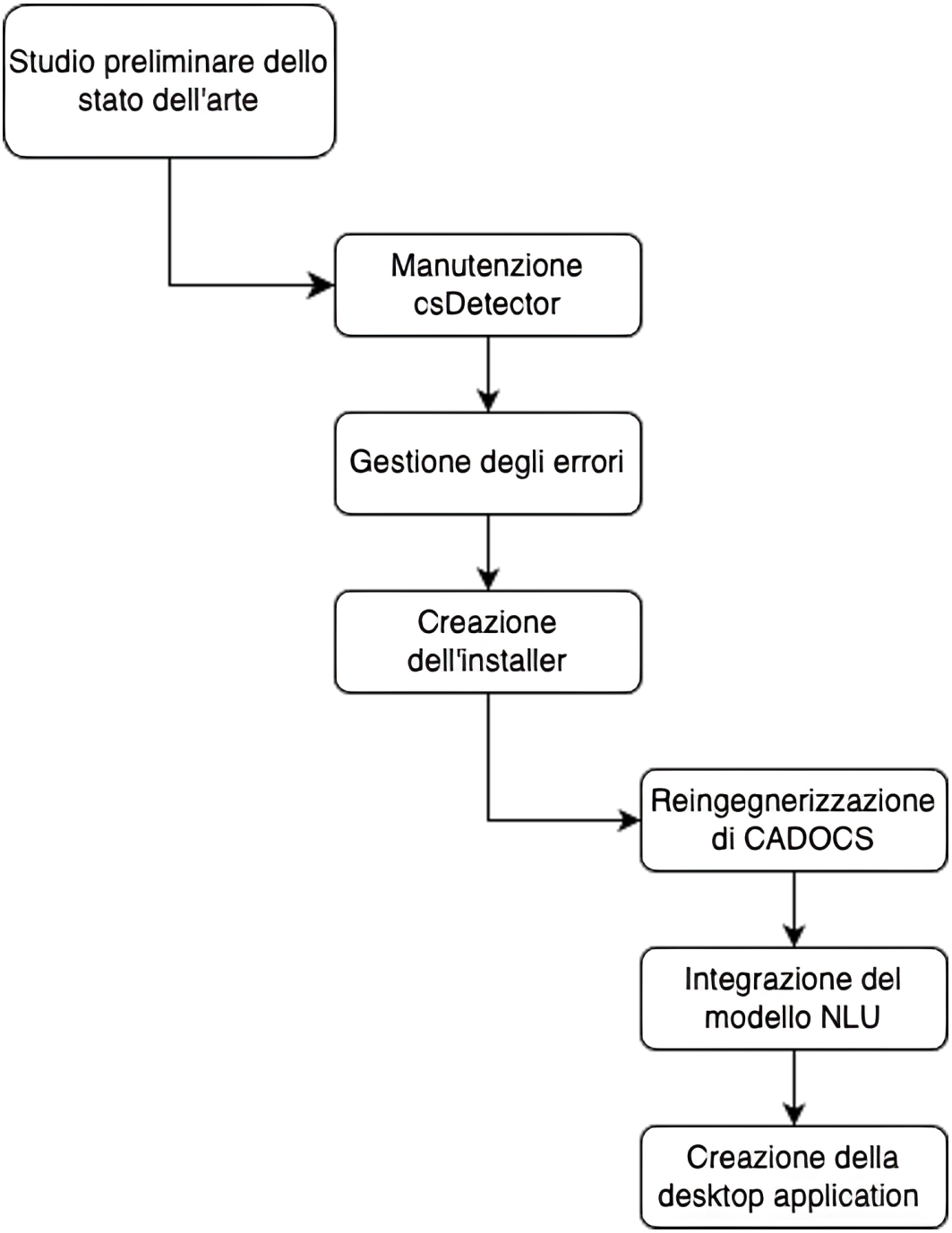

Figura 1.1: Scaletta del nostro lavoro

La rappresentazione schematica di tutte le fasi del lavoro di tesi è illustrata nella Figura 1.1. Questi passaggi hanno guidato lo sviluppo e l'implementazione degli obiettivi chiave di questo progetto di tesi, che sono:

- Gestione degli errori di csDetector. Il tool ha una gestione degli errori qualitativamente bassa (i.e., non si capisce perché fallisce). L'obiettivo è effettuare un lavoro di manutenzione su csDetector.
- csDetector ha un sistema di installazione complesso e lungo, L'obiettivo è rendere il tool accessibile ad un elevato numero di utenti.
- L'obiettivo del lavoro su CADOCS è stato unificare tutti i suoi moduli in un'unica struttura, permettendo al tool di accedere alla maggior parte delle risorse di cui aveva bisogno in locale.
- Il lavoro su CADOCS continua, il secondo nostro obiettivo era trasformare CADOCS in un'applicazione desktop, rendendola accessibile e utilizzabile da chiunque tramite un'installazione semplice sulla propria macchina.

Per raggiungere tali obiettivi, ci siamo concentrati su diverse aree di lavoro. Nel caso di csDetector, ho dedicato tempo a una revisione approfondita del codice per migliorare la gestione degli errori, aggiungendo delle eccezioni *custom* per sollevare le giuste eccezioni quando il tool smetteva di funzionare.

Per il secondo punto ho inoltre ottimizzato il sistema di installazione di csDetector, creando un installer per il tool, rendendolo disponibile ad un pubblico più ampio.

Per quanto riguarda CADOCS, ho lavorato sul merge del modello NLU all'interno dello stesso tool, unendoli in un'unica struttura. Inoltre, ho modificato il tool per trasformarlo in un'applicazione desktop, facilitando così l'installazione e l'utilizzo su diverse macchine, creando così un app che possa funzionare in locale, eliminando quasi del tutto la necessità di richiamare le componenti online, ad eccezione, per esempio, le API per GitHub.

1.5 Struttura della tesi

La tesi è strutturata in 5 capitoli i cui concetti espressi vengono spiegati brevemente di seguito:

- Nel capitolo 2 inizia con una più approfondita analisi di quelli che sono gli studi fatti in ambito Community Smells, viene specificata entrando nel dettaglio

degli smells con i principali lavori specializzati nella rilevazione degli stessi. Infine l'ultima parte del secondo capitolo si concentra più sui chatBot e sulle eventuali appliaczioni che hanno.

- Il capitolo 3 costituisce la prima parte di questo lavoro, concentrandosi sullo sviluppo e sulla manutenzione di csDetector. Viene esaminata l'analisi delle tecnologie considerate e utilizzate per la realizzazione del suo installer, con un successivo sul suo funzionamento.

- Il capitolo 4 rappresenta la seconda parte del lavoro, in questo capitolo, viene effettuata la reingegnerizzazione di CADOCS trasformandolo in un'applicazione desktop e integrando in esso il modulo NLU.

- Il capitolo 5, infine, è quello delle conclusioni, in esso viene fatta una panoramica generale sull'intero progetto. Cosa è andato bene e cosa è andato male. Infine vi è il sottocapitolo sugli sviluppi futuri dove si studiano i miglioramenti possibili che si possono effettuare su questa desktop app.

La repository sia del nuovo csDetector[1], sia di CADOCS 2.0[2] con tutte le indicazioni per usarli, sono disponibili in due repository online.

[1]Link alla repository di csDetector: [https://github.com/PaoloCarmine1201/csDetector]
[2]Link alla repository di CADOCS 2.0: [https://github.com/PaoloCarmine1201/CADOCS_II]

CAPITOLO 2

Background e stato dell'arte

2.1 Community smells

2.1.1 Social debt

Dietro ogni progetto di sviluppo software, è fondamentale garantire che le interazioni tra tutti gli stakeholder siano efficienti e soddisfacenti. In questo contesto, emergono concetti come i *community smells*, che rappresentano segnali di potenziali problemi di comunicazione e collaborazione tra developer che possono influenzare lo sviluppo del progetto. Cosa sono i community smells e a cosa possono portare?

Prima di esaminare approfonditamente i community smells, è essenziale introdurre il concetto di *social debt* (debito sociale). Questo aspetto rappresenta una parte critica nello sviluppo software, ed è stato oggetto di numerosi studi passati.

L'attività di Software Engeneering (SE) è un attività sociale, che comprende un alto coinvolgimento di utenti, come gli sviluppatori, ognuno dei quali può influenzare il lavoro dell'altro con i propri comportamenti. I social debt sono tutti quei costi e i problemi derivanti da una cattiva gestione delle dinamiche sociali e delle relazioni all'interno di una comunità di sviluppo o di un team di programmatori. Questo debito può verificarsi quando ci sono conflitti non risolti, mancanza di comunicazione

efficace [3]. Le ricerche condotte da Tamburri et al. [3, 4] hanno rivelato una stretta correlazione tra il social debt e il *technical debt*, quest'ultimo rappresentando il costo supplementare derivante da pratiche di programmazione che portano a implementazioni di bassa qualità, con conseguente deterioramento complessivo della qualità del codice sorgente.

Inoltre nel loro studio condotto nel 2015 [3], Tamburri et al. hanno condotto una ricerca esplorativa presso grandi aziende, con l'obiettivo di studiare le cause che portano al social debt in contesti pratici. Da questo lavoro è emersa la conclusione che entrambi gli aspetti, sia il social debt che il technical debt, devono essere considerati attentamente in un progetto di gestione del software.

2.1.2 Cosa sono i community smells

Dagli studi condotti da Palomba et al., Tamburri et al., [5, 3, 6] è emersa la definizione dei community smells come caratteristiche socio-tecniche in una comunità di sviluppo, che, sempre secondo la definizione di Palomba et al e Tamburri et al.,[5, 4, 7] possono finire per ostacolare la produzione complessiva del software, compresa la manutenzione e l'evoluzione. Sono comportamenti che se reiterati nel tempo, possono portare al social debt [3], che a loro volta possono trasformarsi in technical debt [3]. Tamburri et al. hanno coniato il termine *community smells* per definire questi comportamenti controproducenti, trattandoli come veri e propri *anti-pattern* comportamentali. La Tabella 2.1 riporta i più comuni community smells identificati in letteratura [6, 8, 9]. I primi community smells trovati erano 9, ne sono stati trovati altri 12 nello studio di Tamburri et al. [6], mentre nel 2019 sempre Tamburri et al. [10] hanno identificato 4 community smell di tipo architetturale, quali: *Lonesome Architecting*, *Obfuscated Architecting*, *Architecting by osmosis* and *Invisible Architecting*.

Community smell	Descrizione
Black cloud	Eccessivo numero di informazioni dovuto alla mancanza di una comunicazione strutturata o di una governance della cooperazione.
Organizational Silo	Aree isolate della comunità di sviluppo che non comunicano, se non attraverso uno o due dei loro rispettivi membri.
Lone Wolf	Sviluppatore sfiduciato che apporta modifiche al codice sorgente senza considerare le opinioni dei suoi colleghi.
Radio Silence	Un membro si interpone in ogni interazione tra le sottocomunità.
Prima donna	Comportamento accondiscendente ripetuto, superiorità, disaccordo costante, non collaborazione da parte di uno o pochi membri.
Priggish Members	Esigere dagli altri un'inutile e precisa conformità o un'esagerata correttezza, soprattutto in modo moraleggiante o irritante.
Code Red	Questo smell identifica un'area di codice così complessa, densa e dipendente da 1-2 manutentori che sono gli unici a poter fare un refactoring.
Unlearning	Un nuovo progresso tecnologico o organizzativo o una best practice che diventa impraticabile se condivisa con i membri più anziani.
Disengagement	Pensare che il prodotto sia abbastanza maturo e inviarlo alle operazioni anche se potrebbe non essere pronto.
Cognitive Distance	Gli sviluppatori percepiscono la distanza a livello fisico, tecnico, sociale e culturale rispetto a coetanei con notevoli differenze di background.

Tabella 2.1: Descrizione dei community smells più comuni

Negli ultimi anni, i community smells hanno attirato sempre più attenzione nel contesto dello sviluppo software. L'individuazione di un elevato numero di community smells all'interno di un progetto è spesso indicativa di problemi all'interno dell'ambiente di lavoro. Risolvere questi problemi può richiedere risorse aggiuntive, ma è fondamentale per mantenere un ambiente di sviluppo sano e produttivo. Rilevare e affrontare i community smells in modo efficace è obbligatorio per garantire la qualità del software e migliorare le dinamiche di lavoro all'interno della comunità di sviluppo. Questo processo aiuta a identificare le aree che richiedono interven-

ti e a concentrare gli sforzi su miglioramenti mirati in determinate aree di lavoro, contribuendo alla creazione di software di alta qualità.

Secondo vari studi, come quello di Almarimi et al. [11], è emerso che una rilevazione efficiente e tempestiva dei community smells può svolgere un ruolo cruciale nell'aiutare gli sviluppatori e i project manager a preservare la salute della comunità di sviluppo. Tale prontezza nell'individuare e affrontare i community smells consente di intervenire prima che la situazione possa degenerare, contribuendo a mantenere un ambiente di lavoro produttivo.

2.2 Tool di rilevazione

2.2.1 CodeFace4Smells

Tra i primi lavori di rilevazione dei community smells abbiamo *CodeFace4Smells*, sviluppato da Tamburri et al. [12], un estensione del tool già precedentemente disponibile *CodeFace* [13] il cui obiettivo principale iniziale di CodeFace4Smells era quello di raccogliere dati relativi a come gli sviluppatori collaborano e comunicano tra loro all'interno di un gruppo o una comunità di lavoro. I ricercatori impiegano CodeFace4Smells per estrarre dati e costruire modelli basati su code smells e community smells. I dati raccolti contengono informazioni sulla gravità dei problemi di codice e alcune misure strutturali del codice stesso.

2.2.2 csDetector

Uno dei lavori più importanti, per quanto riguarda la rilevazione dei community smells, è csDetector [14]. csDetector è un tool basato su precedenti lavori di Almarimi et al. [11], il tool apprende da un insieme di indicatori sociali-tecnici-organizzativi la potenziale esistenza di un community smells, utilizzando il machine learning.

Il funzionamento di csDetector si basa appunto sul machine learning, come prima cosa analizza la cronologia delle modifiche di un progetto software per calcolare diverse metriche sociali e tecniche, che poi saranno utilizzate come caratteristiche o indicatori. Praticamente il programma esamina come il software è stato sviluppato

nel tempo per ottenere dati che verranno successivamente utilizzati come input per ulteriori analisi o calcoli. Il lavoro di csDetector si basa su due tipi di input:

1. csDetector prende come input i modelli di rilevamento dei "community smells" appresi. Questi modelli sono stati precedentemente creati e addestrati per identificare i *community smells* all'interno di un progetto software. Quindi, csDetector utilizza questi modelli per eseguire la rilevazione effettiva degli stessi nel progetto.

2. csDetector prende anche come input la lista delle metriche socio-tecniche estratte dal progetto. Queste metriche rappresentano dati o indicatori relativi alla relazione tra gli aspetti sociali (come la collaborazione tra sviluppatori) e gli aspetti tecnici (come il codice sorgente) del progetto. Le metriche sono state precedentemente estratte dalla cronologia delle modifiche del progetto e serviranno come informazioni utili per il processo di rilevazione.

L'output prodotto da csDetector è la lista delle metriche socio-tecniche che ha calcolato durante l'analisi del progetto, compreso l'elenco dei community smells rilevati. Attualmente csDetector supporta la rilevazione dei community smells più comuni (fare riferimento alla Tabella 2.1). csDetector è stato progettato in modo tale da consentire una facile possibilità di miglioramento ed estensione del tool, in modo tale che gli sviluppatori hanno la flessibilità di adattare o aggiornare i modelli di rilevamento secondo le loro esigenze o requisiti specifici. Praticamente, ora è capace di rilevare i 10 community smells più comuni, ma è possibile sia aumentare il numero di community smells rilevabili che modificare i modelli con cui vengono rilevati quelli già incorporati, nel caso in cui ce ne sia bisogno.

2.2.3 CADOCS

Molte delle soluzioni create per risolvere i community smells nelle comunità di sviluppo erano principalmente progettate per scopi di ricerca, proprio come csDetector [14]. Questi strumenti erano più adatti per condurre studi su larga scala e non tenevano abbastanza conto della facilità d'uso o dell'interesse degli utenti pratici, di conseguenza, molte di queste soluzioni non sono facili da utilizzare nell'ambiente

di lavoro reale, rendendo difficile la loro adozione da parte dei professionisti del settore. Per superare questa limitazione riscontrata negli strumenti precedentemente descritti in letteratura, Voria et al. hanno sviluppato CADOCS (Conversational Agent for the Detection Of Community Smells)[1], un chatbot sviluppato per la piattaforma SLACK, in grado di utilizzare app terze per rilevare community smells tramite repository GitHub.

Come mostrato in Figura 2.1 CADOCS è progettato sulla base di un architettura client-server. Il tool si suddivide in 3 diversi moduli:

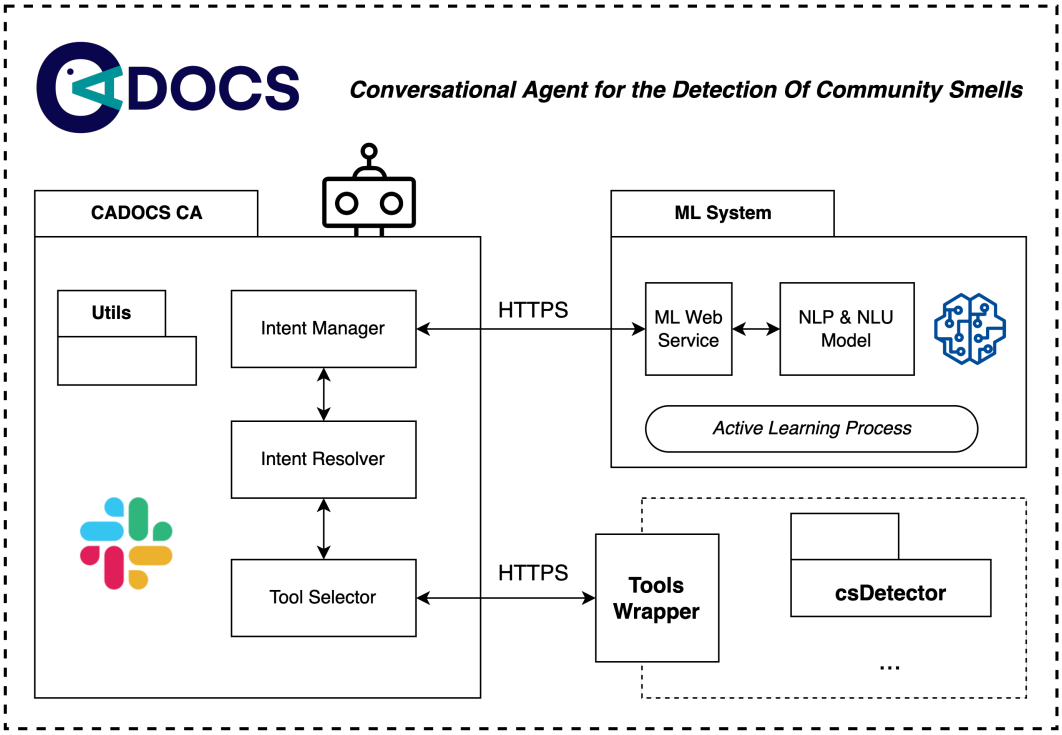

Figura 2.1: Architettura di CADOCS [1]

Tools Wrapper. CADOCS è stato pensato come un tool che wrappa csDetector al proprio interno, fornendo quindi tutte le funzionalità base di csDetector quali: rilevazione dei community smells e misurazione di metriche socio-tecniche. Durante lo sviluppo di CADOCS, Voria et al. hanno affrontato e risolto delle limitazioni presenti in csDetector. Come prima cosa hanno deciso di utilizzare un architettura basata sul web, in modo tale da poterlo utilizzare su ogni dispositivo, senza aver bisogno di un installazione. Inoltre come altra modifica hanno migliorato l'output del tool tramite un organizzazione migliore dei risultati, tramite file CSV e JSON.

Componente NLU. C'è un sistema di Machine Learning dietro al tool, che ha lo scopo di facilitare le risposte in base agli obiettivi dell'utente che pone domande al bot, implementato tramite NLP e NLU. Implementato quindi tramite un modulo che consente di interpretare l'intenzione dell'utente, un modulo ML allenato in base ai modi possibili che hanno gli utenti di chiedere al bot di risolvere la richiesta.

CADOCS CA. La parte principale del tool, contiene al proprio interno la parte che consente lo scambio di informazioni tra l'utente e il tool. Comunica tramite il workspace di *Slack*, implementato proprio tramite le proprie Events API.

CADOCS rappresenta solo uno dei numerosi chatbot implementati per individuare i community smells. L'utilità dei chatbot non è limitata a questa specifica applicazione, poiché offrono numerose possibilità di utilizzo e implementazione. Ma cos'è esattamente un chatbot?

2.3 Chatbot

2.3.1 Presente e passato dei chatbot

Un chatbot è un agente intelligente basato sull'intelligenza artificiale (IA), creato con l'obiettivo di interagire in modo naturale con gli utenti attraverso il linguaggio naturale [15]. Tramite l'IA, questi chatbot sono in grado di comprendere, elaborare e rispondere a richieste, scritte o vocali, dell'utente, utilizzati per svolgere un vasto tipo di operazioni, tra quelle più elementari fino a quelle più laboriose, fornendo un'esperienza di conversazione simile a quella con un essere umano.

Una definizione di chatbot potrebbe essere: *un programma per computer pensato per simulare conversazioni con umani, soprattutto su internet*, è un'entità quanto più simile a un computer, capace di conversare utilizzando linguaggio umano tramite Natural Language Processing (NLP) [16].

Storia. Il termine chatbot è un'abbreviazione del termine originale *chatterbot*, proposto inizialmente nel 1997 da Michael Mauldin, usato per descrivere robot con i quali gli umani potevano comunicare. L'idea di chatbot iniziò a diffondersi con

Alan Turing, matematico ed informatico, nel 1950 con la proposta del Test di Turing, *Possono le macchine pensare?*, un criterio per determinare se una macchina sia in grado di esibire un comportamento intelligente. In breve, il test è una variante dell'esperimento *Imitation game* ed implica la valutazione della capacità di una macchina o di un programma di condurre una conversazione scritta in tempo reale con un essere umano, senza che quest'ultimo riesca a distinguere se sta interagendo con un'altra persona o con una macchina. Lo scopo principale dei chatbot è proprio quello di creare l'illusione nell'utente che stia realmente conversando con un essere umano.

Per comprendere l'evoluzione dei chatbot e il loro impatto, è fondamentale esaminare uno dei primi esempi di intelligenza artificiale applicata alla conversazione umana: *ELIZA* [17]. Sviluppato nel 1966, ELIZA rappresenta un punto di partenza cruciale nella storia dei chatbot, aprendo la strada alla creazione di agenti conversazionali che avrebbero successivamente rivoluzionato il modo in cui interagiamo con le macchine. Utilizzando un sistema di *pattern matching*, ELIZA era un bot senza personalità. *PARRY* [17], sviluppato nel 1972 a solo scopo didattico, è stato il primo vero upgrade di ELIZA. La differenza principale tra i due è che PARRY aveva una propria personalità.

Attualità. Dalla semplicità iniziale di ELIZA e PARRY, pionieri della conversazione artificiale che hanno gettato le basi per ciò che oggi sono i chatbot sofisticati e altamente personalizzati che conosciamo, come Alexa di Amazon, Siri di Apple, Cortana di Microsoft ecc.

Come mostrato in Figura 2.2, l'evoluzione nel campo dei chatbot è stata esponenziale anno dopo anno [18].

2.3.2 Tipi di chatbot ed applicazioni

A differenza di ELIZA e PARRY, i chatbot attuali utilizzano tecnologie avanzate di intelligenza artificiale, come il deep learning e il Natural Language Processing (NLP), per comprendere e rispondere alle esigenze degli utenti in modo più accurato. Come abbiamo già scritto in precedenza, i chatbot stanno assumendo un ruolo sempre più significativo nella società contemporanea, diffondendosi ampiamente e trovando

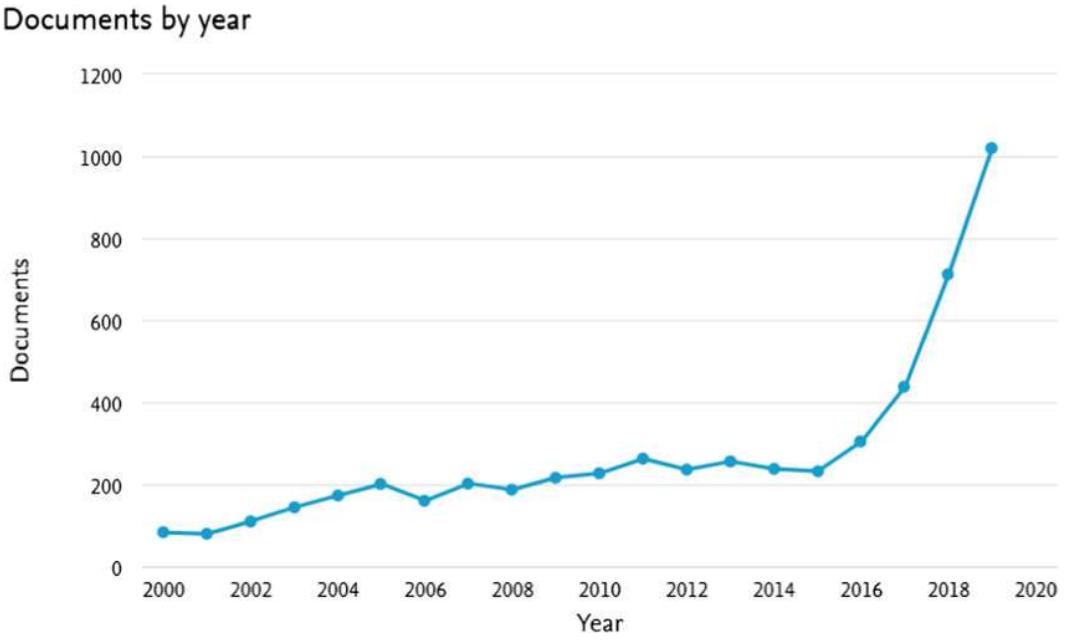

Figura 2.2: Risultati di una ricerca riguardo la crescita del numero di ricerche delle parole chiave *chatbot* e *Conversational Agent* dal 2000 al 2019. [2]

applicazioni in una vasta gamma di attività, come quelle quotidiane, e.g. assistenza clienti, salute, istruzione ecc. Il loro sempre più alto utilizzo da parte delle aziende di maggior rilevanza ha comportato numerosi vantaggi, tra cui risposte immediate, disponibilità 24/7, risparmio di tempo [19].

Quindi principalmente, perchè gli utenti stanno utilizzando sempre di più i chatbot? Il motivo più frequente è la produttività, il loro utilizzo hanno abbassato notevolmente i costi per il servizio clienti ed è aumentata la capacità di gestire più utenti contemporaneamente [20].

Le tecnologie su cui si basano i chatbot sono numerose e di vario tipo:

- **Pattern Matching (ELIZA):** Consiste in un blocco *stimolo-risposta*, dove lo *stimolo* dato in input al chatbot genera una risposta basata sull'input dell'utente. Il principale svantaggio di questa tecnologia è che le risposte sono spesso ripetitive e possono finire in un loop di risposte simili.

- **Artificial Intelligence Markup Language (AIML):** Creato tra il 1995 e il 2000, AIML è una tecnologia che utilizza il pattern matching, ma è applicata a un modello di linguaggio naturale tra due entità, un essere umano e un chatbot.

Utilizza un linguaggio di markup basato su XML con tag di input (<pattern>) e tag di risposta (<template>).

- **Chatscript:** È il successore del linguaggio AIML ed è basato su un linguaggio di scripting open source con un motore di esecuzione. Utilizza regole diverse per ogni macro argomento, cerca l'elemento migliore che corrisponde alla domanda dell'utente ed esegue una regola associata a quel macro argomento. Chatscript include anche una memoria a lungo termine che permette di salvare informazioni importanti dell'utente, come l'età.

- **Natural Language Processing (NLP):** Nell'ambito dell'intelligenza artificiale applicata ai chatbot, NLP si riferisce alla capacità di una macchina di comprendere, interpretare e generare linguaggio umano in modo naturale. Questo permette ai chatbot di interagire con gli utenti utilizzando il linguaggio umano, comprendendo le loro richieste e apprendendo dalle conversazioni precedenti per migliorare le risposte future. Le tecniche di NLP più importanti sono basate sul Machine Learning.

- **Natural Language Understanding (NLU):** È al centro di qualsiasi attività che coinvolge l'uso di NLP. NLU è la capacità di una macchina o di un programma di non solo comprendere il linguaggio umano, ma anche di interpretarne il significato in modo profondo e contestuale. Consente ai chatbot di non solo riconoscere parole e frasi, ma di comprendere il significato implicito dietro le parole, considerando il contesto della conversazione e la semantica delle frasi. Estrapola l'intenzione dell'utente nella sua richiesta e utilizza quella per rispondere in modo appropriato. L'intenzione rappresenta un collegamento tra ciò che l'utente dice e le azioni che il chatbot dovrebbe intraprendere.

Le diverse tecnologie alla base dei chatbot sono fondamentali per la creazione di diverse categorie di chatbot [21]. A seconda degli obiettivi e delle capacità, i chatbot sono divisi in diverse categorie. I chatbot possono essere classificati in base a vari parametri:

- Il dominio di applicazione;

- Il servizio offerto;
- L'obiettivo che devono raggiungere;
- L'input che devono processare e il metodo di generazione della risposta.

Una delle categorie di classificazione dei chatbot si basa sul dominio di applicazione, considerando che i chatbot devono avere accesso a una vasta quantità di dati. In questo contesto, esistono due tipi principali di domini applicativi:

- **Dominio Aperto:** I chatbot di dominio aperto sono progettati per affrontare argomenti generali e rispondere in modo appropriato a una vasta gamma di domande e conversazioni. Questi chatbot hanno la capacità di gestire interazioni su argomenti diversi e di fornire sempre risposte coerenti.
- **Dominio Chiuso:** Al contrario, i chatbot di dominio chiuso sono specificamente progettati per affrontare argomenti ristretti o specifici. Sono in grado di rispondere in modo efficace solo a domande e conversazioni relative a tali argomenti specifici e possono non essere in grado di gestire argomenti al di fuori del loro campo di competenza.

La classificazione basata sul servizio offerto considera quanto un chatbot sia vicino all'utente e quanto il grado di *intimità* influenzi questa classificazione [19]. In questa categoria, esistono diverse tipologie di chatbot:

- **Chatbot Interpersonali:** Questi chatbot rientrano nell'ambito della comunicazione e forniscono servizi come la prenotazione di ristoranti o voli. La loro funzione principale è quella di raccogliere le informazioni richieste dall'utente e inoltrarle senza necessariamente stabilire un rapporto stretto. Non sono concepiti come compagni personali dell'utente.
- **Chatbot Intrapersonali:** Questa categoria di chatbot opera nel dominio strettamente personale dell'utente, come le app di messaggistica istantanea, e.g. WhatsApp. I chatbot intrapersonali cercano di comprendere l'utente il più possibile e interagiscono con lui/lei come farebbe un essere umano. L'obiettivo è creare un'esperienza di conversazione altamente personalizzata e interattiva.

- **Chatbot tra Agenti:** Questo tipo di chatbot si occupa della comunicazione tra chatbot stessi. Sono responsabili dello scambio di informazioni tra diverse entità artificiali. Un esempio concreto potrebbe essere l'integrazione tra due assistenti virtuali come Alexa e Cortana, che comunicano tra di loro per fornire una gamma più ampia di servizi all'utente.

La classificazione in base agli obiettivi dei chatbot prende in considerazione l'obiettivo primario che i chatbot mirano a raggiungere. In questa categoria, ci sono diverse tipologie di chatbot:

- **Chatbot Informativi:** Questi chatbot sono progettati per fornire all'utente informazioni che sono già a loro disposizione, spesso da un database o da una fonte affidabile. Il loro obiettivo principale è quello di rispondere a domande fornendo dati o informazioni specifiche. Un esempio di chatbot informativo sono i bot delle FAQ (Frequently Asked Questions), che forniscono risposte alle domande più comuni degli utenti.

- **Chatbot basati su chat/conversazionali:** Questa categoria di chatbot è progettata per comunicare con l'utente in modo simile a un essere umano, cercando di rispondere correttamente a qualsiasi domanda venga posta nel miglior modo possibile. Questi chatbot mirano a creare un'esperienza di conversazione naturale e sono in grado di gestire conversazioni su una vasta gamma di argomenti. Un esempio è l'assistente vocale di Apple, Siri.

- **Basati su task:** I chatbot in questa categoria sono progettati per svolgere specifici compiti o task. e.g., un chatbot che consente agli utenti di prenotare un volo, effettuare una prenotazione al ristorante. Sono focalizzati sulla risoluzione di problemi degli utenti in un contesto applicativo definito.

La classificazione dei chatbot in base all'input che devono processare e al metodo di generazione della risposta, si basa proprio sul metodo di generazione della risposta. Esistono tre metodologie di generazione:

- **Modello basato sulle regole:** I chatbot con un modello basato sulle regole rappresentano l'architettura più comune nella creazione di un chatbot. In questo

tipo di architettura, le risposte sono determinate in base a un set predefinito di regole, che si basano sul riconoscimento lessicale dell'input fornito dall'utente. Questi chatbot non creano nuove risposte ma selezionano quelle esistenti in base alle regole stabilite. La conoscenza utilizzata in questi tipi di chatbot è inserita manualmente ed è organizzata in base a pattern conversazionali predefiniti [22]. Nella maggior parte delle ricerche relative a questi tipi di chatbot, l'attenzione è focalizzata sulla generazione di risposte a singolo turno, cioè risposte basate esclusivamente sull'ultimo messaggio inviato dall'utente. Tuttavia, nelle risposte a più turni, i chatbot possono somigliare di più a un essere umano, utilizzando più messaggi per generare risposte più rilevanti e precise, tenendo conto dell'intero contesto della conversazione.

- **Modello basato sul recupero:** Questi chatbot sono leggermente diversi rispetto a quelli basati sulle regole, poiché offrono maggiore flessibilità nell'analisi delle domande grazie all'utilizzo di API esterne [23]. Questo tipo di chatbot recupera risposte da un indice di risorse preesistenti prima di selezionare quella che meglio corrisponde alla domanda posta dall'utente.

- **Modello generativo:** I chatbot con un modello generativo sono i più utilizzati tra i tre tipi di chatbot. Questi chatbot generano la risposta utilizzando sia il messaggio corrente dell'utente che quelli precedenti, proprio come farebbe un essere umano. Utilizzano algoritmi di Machine Learning e tecniche di deep learning. Tuttavia, è importante notare che sono più complessi da implementare e richiedono un allenamento più avanzato [23].

CAPITOLO 3

Parte 1: Manutenzione di csDetector

Nel seguente capitolo, si esaminerà il processo di sviluppo del progetto di tesi. L'obiettivo centrale di questa tesi è quello di effettuare un lavoro di manutenzione e evoluzione sull'unico tool di detection dei community smells, csDetector, poi successivamente sul tool che lo rende disponibile, CADOCS. In generale questo lavoro di tesi si è suddiviso in 3 parti principali:

- **Gestione degli errori:** csDetector ha una gestione degli errori qualitativamente molto bassa, tramite exception si è dovuta migliorare tale parte.
- **Integrazione di altri lavori e semplificazione dell'installazione:** Il processo di installazione di csDetector è molto lungo e complesso, l'obiettivo è quello di rendere questo processo il più semplice possibile. In aggiunta, è importante anche l'integrazione di ulteriori lavori, sia in csDetector che in CADOCS, con l'obiettivo di unire tutto in un unico blocco.
- **Trasformare il tool in una Desktop App:** Modificare il tool in modo che possa funzionare nella quasi totalità, in locale, riducendo al minimo la necessità di richiamare altre componenti online. Escludendo alcune componenti che necessitano inevitabilmente di essere richiamate online, come ad esempio le API per GitHub.

3.1 Analisi di csDetector

Obiettivo. Inizialmente, nell'utilizzo precedente del tool, è stato rilevato un problema legato alla gestione degli errori all'interno di csDetector. Questa gestione di bassa qualità ha generato difficoltà nel comprendere le ragioni per cui il tool non eseguiva determinate operazioni durante l'analisi delle repository. Durante l'esecuzione, il tool terminava senza fornire alcun feedback, né all'utente né al programmatore, rendendo difficile da capire su che tipo di analisi il tool si bloccasse. Inoltre, durante un'attenta analisi dello stato dell'arte riguardante le attività di revisione del codice e la documentazione associata a csDetector, è emerso che la documentazione del tool, comprendente la descrizione delle funzionalità, la struttura e l'implementazione, risultava molto limitata e poco informativa.

Da studi precedenti si è notato che è vantaggioso per il programmatore sfruttare principalmente i vantaggi che offre l'analisi statica, naturalmente l'analisi del codice non deve limitarsi solo a queesto tipo di lavoro ma il modo migliore per certificare che un'implementazione abbia il minor numero possibile di errori o difetti è combinare sia misure statiche che dinamiche di analisi, poiché ciascuna fornisce un insieme diverso di informazioni e può rivelare difetti o errori che l'altra potrebbe non individuare.

3.1.1 Analisi statica

Ci siamo concentrati inizialmente sull'analisi statica approfondita del codice sorgente di csDetector, accompagnata dalla comprensione approfondita del suo funzionamento, tra documentazione e codice stesso. Una fase preliminare che ci è stata fondamentale acquisire una panoramica completa del progetto in questione.

Abbiamo iniziato ad analizzare, per ogni package, il funzionamento di ogni singola classe al proprio interno, cercando di comprendere approfonditamente tutti i processi e le operazioni eseguite da essa. Durante questo lavoro, abbiamo esaminato in modo dettagliato il flusso di lavoro dell'applicazione, le sue caratteristiche fondamentali cercando di comprendere la logica dietro a ogni operazione eseguita.

La ricerca iniziale si è focalizzata soltanto alla comprensione del funzionamento *teorico* del tool, poi successivamente, ci siamo concentrati sull'identificazione di possibili errori, eccezioni non gestite o eventuali malfunzionamenti presenti. Analizzando

tutte le possibili porzioni di codice che avrebbero potuto generare questi tipi di problemi. L'obiettivo di questa analisi è stato, quindi, quello di cercare di individuare quanti più problemi nel codice senza richiedere l'attivazione del tool. Cercando di identificare tutte le sue aree che avrebbero richiesto poi un attenzione più specifica, attenzione che avrà durante l'analisi dinamica e che richiederà quindi l'utilizzo vero e proprio del tool.

3.1.2 Analisi dinamica

L'analisi del comportamento dinamico del codice è molto importante per gli sviluppatori poiché aiuta a comprendere a fondo il software stesso. L'analisi statica non richiede l'esecuzione del programma, quindi, non fornisce quei dati che solo l'esecuzione di un programma può fornire.

Questo tipo di analisi serve quindi per comprendere il comportamento di un software durante l'esecuzione. L'analisi dinamica raccoglie informazioni sullo stato del programma mentre viene eseguito, registrando dati come le chiamate alle funzioni, le variabili utilizzate e le istruzioni eseguite. Questa tecnica consente di individuare problemi di esecuzione o comportamenti anomali nel software. Si può eseguire tramite il monitoraggio in tempo reale dell'esecuzione del programma o analizzando i dati raccolti dopo l'esecuzione.

L'analisi si è basata, oltre che su un dataset di repository *GitHub Open Source*, anche su un dataset di repository degli studenti del corso di *Ingegneria del Software* dell'anno accademico 2022/2023. Una volta eseguito il tool per ogni repository, abbiamo tenuto traccia degli eventuali errori e malfunzionamenti che venivano restituiti in output per ogni esecuzione. La Tabella 3.1 riporta alcune delle repository sulla quale è stato testato il tool.

Nome progetto	Repository
BeeHave	https://github.com/gianwario/BeeHave.git
Report.it	https://github.com/simonagrieco/Report.it.git
EnIA	https://github.com/benedettoscala/EnIA.git
Fit Diary	https://github.com/fasanosalvatore/FitDiary.git
Comun-Ity	https://github.com/andreaceto/Comun-ity.git
HeartCare	https://github.com/DinoDx/HeartCare.git
Tensorflow	https://github.com/tensorflow/ranking.git
Organic-Shop	https://github.com/ritwickdey/organic-shop.git
Uni-Segnala	https://github.com/alessaless/UniSegnala.git
Gitmoji	https://github.com/carloscuesta/gitmoji.git

Tabella 3.1: Alcune delle repository di test

3.2 Problemi di csDetector

Prima e durante l'esecuzione di csDetector si è andati in contro a diversi tipi di problematiche ed errori:

- **Errori di esecuzione** Errori durante l'analisi dinamica vera e propria del tool. Il tool è stato testato sia a *linea di comando* che come *webService*
- **Errori di installazione** Errori riscontrati durante i vari tentativi di installazione del tool, una serie di errori dovuti al processo particolarmente complesso.

3.2.1 Errori di esecuzione

Dopo aver testato il tool su un totale di quasi 60 repository, trovando un totale di 7 errori, gli stessi sono stati catalogati in base al tipo di errore:

- **Errori di programmazione/merge** Errori che derivano dalla mancata verifica di specifiche variabili, o dovuti dall'assenza di controlli su di esse. Oppure potrebbero essere errori che sono emersi a causa di un merge sbagliato di branch su GitHub, che hanno generato dei conflitti sul codice sorgente.

- **Mancanza di eccezioni** Errori che si verificano durante determinate analisi di metriche nell'esecuzione del programma, dovuti alla mancata gestione di eccezioni personalizzate. Il software presentava un comportamento in cui, in caso di errore, terminava l'esecuzione sollevando un'eccezione generica senza fornire all'utente informazioni dettagliate sulla causa dell'errore.

È stata creata inoltre una Tabella per monitorare i progressi nella risoluzione dei 7 errori individuati, aggiornandola man mano mentre si lavorava alla loro soluzione. La Tabella 3.2 gli riporta gli errori individuati.

idTest	Descrizione	Input
Download1	devNetwork - ERROR - Cmd('git') failed due to: exit code(128) cmdline: git clone –branch=master –progress -v	https://github.com /fasanosalvato-re/FitDiary
Commit1	devNetwork – ERROR – name 'endDate' is not defined	https://github.com/ gianwario/BeeHave
DrawGraph1	devNetwork - ERROR - addToSmellsDataset() takes 3 positional arguments but 4 were given	https://github.com/ gianwario/BeeHave
DrawGraph2	devNetwork - ERROR - mean requires at least one data point	https://github.com/ gianwario/BeeHave
Request1	devNetwork - ERROR - list index out of range	https://github.com/ benedettoscala/EnIA
Request2	devNetwork - ERROR - list index out of range	https://github.com/ ritwickdey/organic-shop
Utterance1	devNetwork - ERROR - 'id'	https://github.com/ DinoDx/HeartCare

Tabella 3.2: Errori trovati nel testing di csDetector

3.2.2 Errori di installazione

Oltre agli errori di esecuzione precedentemente specificati, sono stati incontrati una serie di errori anche durante l'installazione del tool, affrontandone diversi che hanno reso il processo di installazione particolarmente complesso.

Per l'installazione e l'esecuzione del tool, è necessario disporre di diversi requisiti. Tra questi ci devono essere specifiche librerie o dipendenze software, un'adeguata configurazione dell'ambiente di sviluppo e diversi servizi esterni. Questa varietà di prerequisiti può rendere il processo di installazione un procedimento adatto soltanto a utenti esperti, in grado di configurare in modo appropriato gli ambienti di sviluppo e le dipendenze necessarie.

Nello specifico, l'installazione del tool ha bisogno di questi diversi passaggi prima di poterlo utilizzare:

- **Clonazione** Clone della repository in locale in una determinata directory.
- **Requisiti di sistema:**
  - *Windows 10*
  - *Python 3.8**
  - *Java 8.**
- **Requisiti di ambiente:** come l'installazione del file *requirements.txt*, l'installazione di convoKit, un nuovo toolkit che contiene strumenti e metodi ideali per lo sviluppo e la formazione di modelli PNL progettati per analizzare conversazioni umane [24]. Per la precisione bisogna installare *SpaCy*, una libreria Open Source di convoKit.
- **Installazione di Java** Il tool richiede l'installazione di una versione specifica di Java. Inoltre, necessita di fare il *download* di una cartella dati denominata *SentiStrenght*, insieme a un file **.jar* associato, entrambi essenziali per eseguire l'analisi del sentimento [1] di ciascun messaggio che viene poi analizzato.

[1] determina il tono emozionale dietro le parole, per capire le attitudini e le opinioni che vengono espresse.

- Come ultima cosa ha bisogno di due directory denominate *out*, utilizzata per l'output che il tool fornisce all'utente, e *SentiStrenght*, dove vanno inseriti la cartella di dati e il file .jar precedentemente scaricati.

L'insieme di tutti questi prerequisiti quindi, come già anticipato in precedenza, aggiunge complessità al processo di installazione del tool, rendendola una procedura molto impegnativa e soggetta a diversi possibili errori, risultando una procedura noiosa e scoraggiante soprattutto per gli utenti meno esperti con tali requisiti.

3.3 Soluzione proposta

Durante il processo di risoluzione degli errori e delle problematiche riscontrate, sono state identificate delle soluzioni che hanno contribuito a superare i problemi avuti. Queste soluzioni sono state adottate per affrontare specificamente le difficoltà sia legate agli errori di esecuzione che alle problematiche dell'installazione. Per garantire il corretto funzionamento e ridurre potenziali errori, è stato seguito un approccio metodico, eseguito varie prove e adottato strategie specifiche per ciascun problema individuato. Sono state proposte delle soluzioni quindi per entrambe le macro problematiche affrontate, errori di esecuzione e errori di installazione.

3.3.1 Gestione delle eccezioni

Prima di affrontare la risoluzione dei problemi di esecuzione, è stata intrapresa un'analisi preliminare per comprendere il motivo per cui certe operazioni venivano eseguite e l'importanza di queste azioni nel contesto del processo complessivo.

C'era bisogno di determinare se l'interruzione dell'esecuzione per merito di un'operazione specifica non andata a buon fine, fosse un comportamento necessario o fosse invece un errore di programmazione. Era essenziale comprendere se il tool quindi dovesse terminare l'esecuzione in quel punto o se fosse necessario evitare tale interruzione per garantire il corretto svolgimento delle operazioni successive.

Questo approccio ha richiesto una valutazione dettagliata delle classi che erano coinvolte in quel determinato errore, non solo per identificare le operazioni stesse, ma anche per comprendere la loro relazione con il funzionamento globale del sistema.

Attraverso un'indagine approfondita, abbiamo cercato di individuare le cause dietro ogni problema di esecuzione, analizzando le interazioni tra le varie classi e identificando le possibili interruzioni del flusso operativo. Questo processo di analisi dettagliata è servito a identificare quindi le aree critiche che necessitavano di correzioni, a delineare una strategia mirata per risolvere efficacemente i problemi riscontrati e, Allo stesso tempo, ha chiarito anche le aree era appropriato che il tool terminasse la sua esecuzione, quindi aree che non necessitavano del mio intervento.

Il nostro processo è iniziato, innanzitutto, trovando le repository che restituivano determinati errori, come visualizzato dalla Tabella 3.2. Una volta trovati gli errori, stampati dal log, è iniziato il processo di analisi del codice.

Il processo, esaminando ogni repository, si divideva in diversi step. Il primo passo è stato quello di identificare, prima di tutto il file in cui terminava l'esecuzione il programma, poi successivamente la funzione/le funzioni responsabili dell'errore.

Successivamente è stato sfruttato il debugger di *visual Studio Code* per analizzare nel dettaglio il flusso di esecuzione del codice. Abbiamo usato il debugger perchè era il processo più veloce per l'individuazione del punto esatto in cui veniva lanciato l'errore. Questo approccio mirava a comprendere le variabili coinvolte, le istruzioni in esecuzione e le interazioni tra i diversi componenti del programma.

Dopo numerosi test, una volta individuato il punto critico, è stato avviato il processo di correzione, esaminando prima di tutto, come anticipato in precedenza, se quel determinato errore era:

- **Critico**, il programma deve concludere le analisi in corso. Nel caso di errori di questo tipo, quando sono stati individuati, si è notato un comune denominatore: il tool terminava a causa di liste di oggetti vuote. Le funzioni restituivano queste liste vuote, e il tentativo di operare su di esse causava l'interruzione del programma.

  Per affrontare questa problematica, è stato introdotta una soluzione in csDetector: abbiamo implementato una classe *customException*. Quando sollevata, questa eccezione stampa nel log di sistema dettagli sulla causa dell'errore, identifica la lista vuota coinvolta e genera un codice univoco che agevola il riconoscimento dell'eccezione personalizzata.

- **Secondario**, il programma non deve interrompere la sua esecuzione; è necessario garantire il proseguimento delle operazioni, ad esempio assegnando specifici valori di default alle variabili coinvolte. Una volta individuati errori di questo tipo, abbiamo notato che si trattava, per la maggior parte, di errori di merge o di controlli mancanti non gestiti. Una volta risolti, il tool è stato in grado di proseguire con la sua esecuzione senza problemi.

La Tabella 3.3 riporta un elenco dettagliato degli errori, ognuno identificato da un ID seguito da note esplicative e eventuali proposte di soluzione per ciascun problema.

idTest	Note	Soluzione
Download1	La repository fornita come input genera un errore durante il download poiché cerca di recuperare la repo, tramite il comando *git clone*, dal branch denominato *master*, mentre la repository ha il branch principale denominato *main*	File *'repoLoader'* Funzione *'getRepo'* Cambiare valore all'attributo *'branch'*
Commit1	CsDetector cerca di eseguire il batch dei commit, ma durante il processo, la struttura dati 'endDate', che tiene traccia della data dei commit, non è stata inizializzata correttamente	File *'commitAnalysis'* Funzione *'commitAnalysis'* Assegnare valore alla variabile *'EndDate'*
DrawGraph1	La repository data in input produce un errore durante l'esecuzione della funzione che aggiunge i community smells al dataset, gli viene passato un argomento in più rispetto a quelli che chiede la funzione. L'argomento in questione è il path del file excel nella quale stampare i community smells	File *'devNetwork'* Funzione *'addToSmellDataset'* elimino argomento *'./communitySmellsDataset.xlsx'*
DrawGraph2	La repository fornita in input produce un errore durante il calcolo della lunghezza media dei commenti relativi a un problema (issue) e quelli relativi a una pull request, commenti salvati nelle liste denominate *'issueCommentBatch'* e *'prCommentLenghts'*	File politenessAnalysis funzione calculateAccl controllo se issueCommentBatches oppure prCommentLengths vuoto allora mettiamo le rispettive medie a zero altrimenti vengono calcolate
Request1	La repository data in input produce un errore durante la richiesta per i dati a gitHub. La richiesta restituisce un oggetto vuoto e il programma itera su quell'oggetto e da errore	File *'issueAnalysis'* Funzione *'issueRequest'* Lancio un eccezione: if not nodes: raise Exception("Error: The response from runGraphqlRequest is empty.")
Request2	La repository data in input produce un errore durante l'analisi dei commenti di una pullRequest in un determinato batch, la lista di commenti è vuota ed iterando su quell'oggetto da errore	File *'devNetwork'* Funzione *'devNetwork'* Lancio un eccezione perché non ci sono commenti nella pullRequest if(len(prCommentBatches) < 1): raise Exception("ERROR, There are no comments in pullRequest")
Utterance1	La repository data in input produce un errore durante la costruzione della lista delle utterances. Non ci sono commenti nelle pull request della repo, quindi viene restituita una lista di utterances vuota	File *'PolitenessAnalysis'* Funzione *'getResult'* Lancio un eccezione: if not utterances: raise Exception("Error: There are no comments in the repository, The utterances are empty.")

Tabella 3.3: Tabella con le eventuali soluzioni degli errori

3.3.2 Creazione dell'installer

Per quanto riguarda i problemi di installazione, se ne presentavano di vario tipo, da quelli legati alle dipendenze mancanti o non correttamente configurate, fino a quelli causati da conflitti di versione tra librerie. Alcuni casi richiedevano soltanto una revisione dei prerequisiti di sistema, mentre altri necessitavano di correzioni più approfondite nell'ambiente di sviluppo. Queste diversità di errori richiedeva una risoluzione caso per caso.

I problemi principali incontrati durante l'installazione erano:

- La corretta creazione dell'ambiente virtuale, con tutte le dipendenze associate;
- La corretta creazione delle directory 'output' e 'SentiStrenght'
- Il corretto avvio del webService in modo tale da avere le giuste risposte.

Per affrontare tali problematiche, abbiamo optato per l'implementazione di un *csDetectorInstaller*. Questo tool ha reso l'installazione di csDetector un processo più agevole, eliminando la necessità di operazioni complesse da riga di comando e rendendo superfluo il download manuale di librerie esterne. Inoltre, ha fornito una maggiore chiarezza nel processo di installazione, riducendo le possibili confusioni derivanti da questi passaggi noiosi.

3.3.3 Tecnologie usate

Abbiamo optato per l'utilizzo di *Python*, la stessa tecnologia impiegata per csDetector, per sviluppare l'installer. Inoltre, abbiamo creato un'interfaccia grafica abbastanza basilare, utilizzando la libreria TkInter di Python, per rendere così il processo di installazione ancora più semplice.

Python, linguaggio che, per quanto semplice, presenta un numero di librerire che lo rendono unico nel suo genere. In questo caso è stata utilizzata la libreria TkInter, libreria Python che fornisce una serie di strumenti per creare interfacce utente grafiche (GUI). Tkinter si basa sul toolkit *GUI TCL/TK*, che fornisce i componenti grafici sottostanti. Consente quindi agli sviluppatori di utilizzare la sintassi di Python per creare e manipolare i componenti della GUI. Le caratteristiche principali di Tkinter

includono una vasta gamma di widget GUI, tra cui pulsanti, etichette e caselle di testo, la gestione del layout, con layout manager per organizzare i widget sullo schermo, e la gestione degli eventi, per rispondere alle azioni degli utenti tramite funzioni di callback.

3.3.4 Funzionalità

Come accennato in precedenza, l'interfaccia è stata progettata con semplicità, mirando a essere accessibile a tutti gli utenti senza richiedere passaggi complessi.

La Figura 3.1 mostra la GUI dell'installer. Oltre l'interfaccia semplificata, il suo funzionamento è altrettanto semplificato, consentendo di automatizzare diverse operazioni tramite pochi pulsanti:

- **Seleziona directory**, un pulsante che se cliccato, consente di selezionare la cartella in cui si desidera clonare la repository di csDetector tramite l'installer. Questo processo è reso possibile grazie alla classe chiamata *FileDialog* di TkInter, che apre il *Finder* della tua macchina, permettendoti di esplorare e selezionare la cartella di destinazione

- **Clona repository**, un pulsante che se cliccato, effettua varie operazioni. Tramite l'utilizzo della libreria *Subprocess*[2] di Python, l'installer apre un terminale di sistema in background e avvia una serie di comandi:

  - consente di clonare la repository nella cartella selezionata al passo precedente, tramite il comando *git clone https://github.com/PaoloCarmine1201/csDetector.git*.
  - crea le cartelle 'output' e 'SentiStrenght' all'interno della repository appena creata, tramite le istruzioni *os.makedirs*.
  - crea l'ambiente virtuale con la versione di python 3.8 all'interno della repository, tramite il comando *python3.8 -m venv .venv*.

- *Start virtual env*, un pulsante che se cliccato, prima di procedere, verifica se l'ambiente attuale è attivo. Nel caso non lo fosse, esegue vari comandi:

[2]Libreria di Python che offre la possibilità di creare e lavorare con processi addizionali, e.g. aprire il terminale di sistema

  - attiva l'ambiente virtuale tramite il comando *source .venv/bin/activate*
  - installa tutte le dipendenze necessarie per l'utilizzo del tool tramite il comando *pip install -r requirements.txt*
  - installa la libreria open source di ConvoKit SpaCy tramite il comando *python3 -m spacy download en_core_web_sm*

- *Start webService*, un pulsante che se cliccato, anche in questo caso, prima di procedere, verifica se l'ambiente attuale è attivo. Nel caso non lo fosse lo attiva e avvia il webService tramite il comando *python csDetectorWebService.py*

Figura 3.1: GUI dell'installer di csDetector

Una volta installato csDetector in locale sulla macchina tramite questo installer, il tool per essere utilizzato ha bisgono soltanto di due requisiti, dopo aver effettuato queste due operazioni da effettuare manualmente, il tool è pronto per l'utilizzo:

- una qualsiasi versione di Python 3.8 installata sulla macchina.
- l'inserimento all'interno della cartella denominata 'SentiStrenght' creata dal tool, del file .jar e del file dati di SentiStrenght.

CAPITOLO 4

Parte 2: Reingegnerizzazione di CADOCS

4.1 Obiettivo e motivazioni

In questo capitolo vengono mostrate le scelte di design e il processo di sviluppo delle modifiche da apportare a CADOCS. Come specificato in precedenza, CADOCS nasce come un conversational agent che opera su *Slack*, wrappando il tool csDetector ed utilizzandolo come webService per la detection di community smells.

Lo scopo iniziale di questo lavoro di tesi è quello di rendere CADOCS una desktop app in modo che possa funzionare in locale, eliminando quasi del tutto la necessità di richiamare le componenti online, ad eccezione, per esempio, le API per GitHub.

4.1.1 Analisi preliminare

CADOCS, come illustrato nella precedente Figura2.1, è un tool che si basa su tre componenti principali:

- Tools Wrapper
- Modello NLU
- CADOCS CA

Con questa architettura, CADOCS non è concepito per funzionare in locale. Ad ogni richiesta inviata, il tool prima chiama il modulo NLU per ottenere l'intent della richiesta dell'utente. Successivamente, utilizzando la risposta del modulo NLU, viene effettuata una richiesta al web service di csDetector per individuare i community smells, se necessario. L'obiettivo principale si divide in due punti:

- Includere il modello NLU all'interno del core vero e proprio dell'applicazione.
- Implementare una comunicazione diretta tra CADOCS e csDetector, evitando l'obbligo di utilizzo del web service come intermediario.

Dopo un'approfondita analisi delle tecnologie disponibili e delle architetture dei due tool, si è tratti a una conclusione. Nel primo caso, l'inclusione non ha presentato problematiche significative, mentre per quanto riguarda il secondo punto, sono emersi diversi inconvenienti.

4.1.2 Comunicazione tra i due tool

Nel corso dello sviluppo del tool, come accennato in precedenza, l'obiettivo principale è stato quello di stabilire la fattibilità di una comunicazione efficace tra CADOCS e csDetector. Questo è risultato essere il fulcro centrale delle problematiche incontrate durante il processo di sviluppo.

csDetector è stato concepito su una qualsiasi versione di Python 3.8.*, con determinate dipendenze studiate appositamente per renderli compatibili con la stessa. Invece CADOCS è stato creato per funzionare su una qualsiasi versione di Python 3.9, anche in questo caso, con le proprie dipendenze compatibili con quella versione del linguaggio di programmazione.

La decisione di abbandonare la migrazione di csDetector a una versione successiva di Python è stata presa a causa delle complessità elevata e dei tanti problem che avrebbe comportato tale scelta. Poiché i due tool richiedevano ambienti virtuali distinti per funzionare correttamente, bisognava condurre diversi test al fine di verificare la possibilità di comunicazione tra questi diversi ambienti, in modo tale da eliminare la dipendenza dal webService di csDetector.

Dopo una serie di approfonditi test finalizzati a consentire la comunicazione tra i due *virtual environment* dei rispettivi tool, abbiamo optato per non perseguire questa soluzione implementativa e continuando a fare affidamento sul webService. La decisione di non proseguire con la comunicazione diretta tra i due ambienti virtuali è stata presa a causa della complessità nel rendere interoperabili tali ambienti. È risultato difficile stabilire una comunicazione efficace tra di essi, richiedendo notevoli sforzi senza garantire un'implementazione efficace. Di conseguenza, abbiamo ritenuto più pratico e ragionevole continuare a fare affidamento sul webService di csDetector. Tuttavia, sono stati dedicati sforzi a semplificare l'installazione di csDetector direttamente sulla macchina locale, motivo per cui si è scelto di sviluppare un apposito installer per questo tool.

4.1.3 Funzionalità chiave

Una volta chiarite le problematiche legate alla comunicazione tra ambienti virtuali, ed aver optato per l'implementazione dell'installer, ci siamo concentrati sulla definizione delle funzionalità principali che il nuovo CADOCS dovrà avere. Questo passaggio cruciale è stato pensato ponendo un'enfasi particolare sull'integrazione ottimale del modello NLU all'interno del nuovo tool. Ecco un elenco delle principali funzionalità che il nuovo CADOCS deve offrire:

1. **Facile installazione:** l'obiettivo più importante che deve avere il tool è quello di poter essere utilizzato senza lunghi e particolari processi di installazione.

2. **Integrazione del modello NLU:** era importante avere disponibile all'interno del tool il modello NLU, unito al modello di riconoscimento della lingua.

3. **Login:** una nuova funzionalità introdotta rispetto a CADOCS è l'aggiunta di una pagina di login tramite GitHub, che consente l'accesso esclusivamente attraverso questa piattaforma.

4. **Gestione delle eccezioni:** facendo riferimento all'eccezione personalizzata creata per csDetector, una volta sollevata quel tipo di eccezione era importante catturarla e gestirla anche nel nuovo tool.

4.2 Struttura generale

In questa sezione verrà trattato sia il linguaggio di programmazione utilizzato durante lo sviluppo, che delle librerie di supporto che sono state valutate e utilizzate per la realizzazione del tool.

4.2.1 Tecnologie utilizzate

In particolare per quanto riguarda i linguaggi di programmazione è stato preferito completare il progetto in Python grazie alle sue librerie molto più facili da utilizzare e, soprattutto, più complete e versatili rispetto a quelle disponibili in altri linguaggi.

Il nuovo CADOCS è stato progettato e sviluppato con Python 3.11, una scelta strategica mirata anche a integrare ottimamente il modello NLU. Utilizzando la stessa versione di Python, siamo riusciti a mitigare i potenziali problemi di fusione tra il tool e il modello. Questa scelta ha agevolato il processo di merge, richiedendo solamente una breve sessione di refactoring del codice, senza incontrare complicazioni.

Per quanto riguarda, invece, le tecnologie utilizzate per sviluppare tutte le principali funzionalità della desktop application sono:

- **Modulo per l'installazine:** Per fare in modo che CADOCS sia eseguibile sulla macchina in locale abbiamo deciso di utilizzare *Pyinstaller*.

  Pyinstaller è una libreria Python che viene utilizzato per impacchettare il codice Python in applicazioni eseguibili autonome per vari sistemi operativi. Legge uno script, analizza il codice per scoprire ogni altro modulo e libreria di cui il tuo script ha bisogno per essere eseguito, Quindi raccoglie copie di tutti quei file e li inserisce all'interno del file .exe che successivamente crea, incluso l'interprete attivo Python.

  Ho scelto di utilizzare Pyinstaller perchè è una libreria molto potente, prende uno script Python e genera un singolo file eseguibile che contiene tutte le dipendenze necessarie e può essere eseguito su computer che non hanno il linguaggio installato. Ciò consente una facile distribuzione delle applicazioni Python, poiché l'utente non ha bisogno di averlo installato, insieme a tutti i moduli richiesti sul proprio sistema per eseguire l'applicazione. Inoltre, PyIn-

staller può anche essere utilizzato per creare eseguibili a un file, che sono singoli file eseguibili che contengono tutte le dipendenze richieste per l'applicazione. Ciò può rendere ancora più semplice la distribuzione dell'applicazione, poiché l'utente deve solo scaricare un singolo file.

- **Implementazione della GUI:** Per implementare la GUI della desktop application è stato scelto di utilizzare, per rimanere coerenti con la scelta effettuata per implementare l'installer di csDetector, la libreria TkInter. Insieme a questa libreria, abbiamo utilizzato il modulo aggiuntivo *ttk*. Questo modulo mira a separare il codice che gestisce il comportamento di un widget da quello che definisce l'aspetto estetico, offrendo così una maggiore modularità nel codice.

- **Sistema di login:** Il processo di login è stato progettato utilizzando delle richieste alla API di GitHub, che tramite da una pagina, offre la possibilità di accedere proprio tramite il proprio account GitHub. Una volta selezionato il pulsante di accesso, l'utente riceve un *PAT* [1], una volta ricevuto, il sistema reindirizza automaticamente verso la pagina principale, consentendo di porre domande al chatbot. La descrizione dettagliata del suo funzionamento verrà effettuata più avanti all'interno del capitolo.

4.2.2 Architettura del sistema

L'architettura selezionata per la realizzazione dell'estensione è di tipo client-server, la stessa architettura originale di CADOCS ma con quale modifica. Il sistema viene diviso in tre moduli:

- **ToolGUI**, la parte principale del tool, contiene la logica che interagisce con l'utente, La principale differenza rispetto alla versione precedente è l'inclusione del modello di NLU nella stessa,

- **LoginGUI**, questo modulo gestisce il processo di login, inoltrando richieste HTTP a GitHub per ottenere il codice e autorizzare il tool.

[1] Personal Access Token, usato come password alternativa per l'autenticazione a GitHub, quando si usa l'API

- **Tools Wrapper**, modulo che rimane inalterato rispetto alla prima versione di CADOCS, mantiene tutte le funzioni che venivano offerte precedentemente.

La Figura sottostante (4.1) mostra il diagramma architetturale del tool:

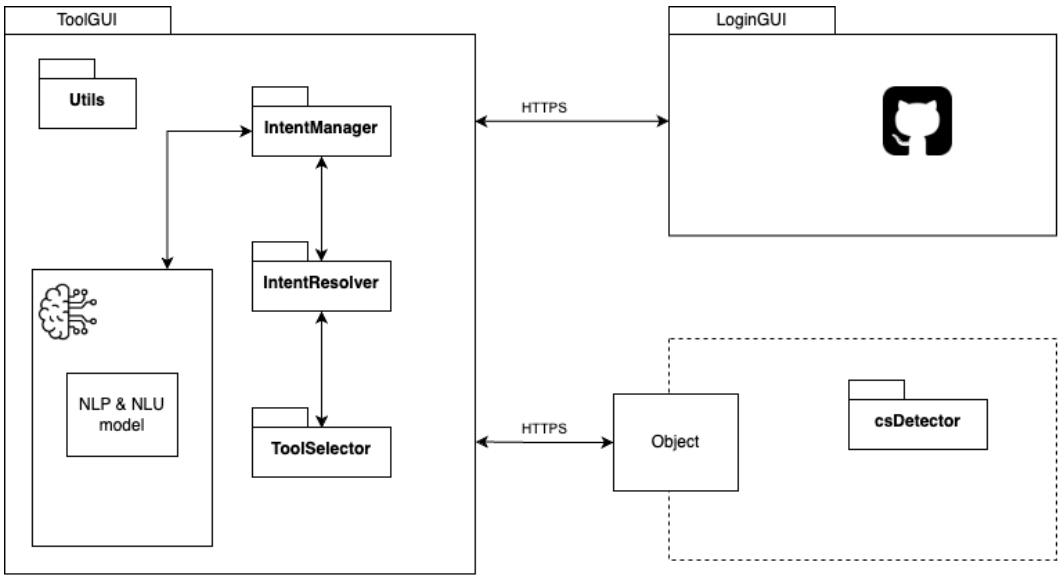

Figura 4.1: Architettura aggiornata di CADOCS

4.3 Implementazione

In questa sezione, si presenta dettagliatamente il sistema, comprendente una panoramica della sua interfaccia e un'analisi del suo funzionamento. Sarà presentata una descrizione delle diverse funzionalità offerte e di come queste operano per il corretto funzionamento generale dell'applicazione.

4.3.1 Pagina di Login

La pagina di login rappresenta la prima che viene visualizzata dopo l'avvio dell'applicazione, per accedere ed utilizzare CADOCS. La pagina offre un interfaccia molto semplice ed intuitiva per l'autenticazione degli utenti. Tramite l'integrazione con GitHub, consente agli utenti con un account di accedere al sistema garantendo un'esperienza d'accesso sicura. Il processo di login è suddiviso in due step:

1. Una volta che si preme il pulsante *Login* nella schermata iniziale (facendo riferimento alla Figura 4.2), l'utente viene reindirizzato alla pagina successiva.

Qui troverà istruzioni dettagliate su come procedere con l'accesso, insieme a un codice specifico. È necessario copiare questo codice e inserirlo nella pagina di autenticazione di GitHub per completare la procedura di login.

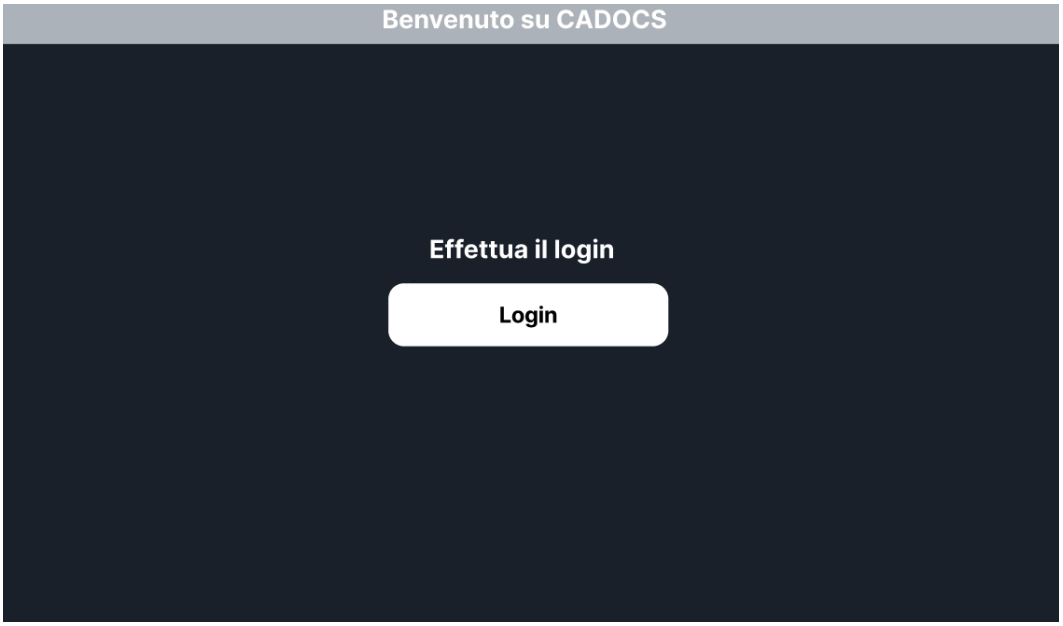

Figura 4.2: GUI iniziale di login

2. Una volta copiato il codice univoco di GitHub, tramite il modulo *webbrowser*[2] di Python viene aperto in automatico la pagina, dove bisogna inserire il codice univoco fornito precedentemente. Una volta inserito il codice l'utente potrà finalizzare il processo di login, autorizzando CADOCS ad utilizzare il proprio account GitHub. Dopo aver dato le giuste autorizzazioni può tornare alla pagina successiva di login (facendo riferimento alla Figura 4.3) e, una volta cliccato su *Confirm login*, l'utente viene reindirizzato alla pagina principale di CADOCS per iniziare a utilizzare il chatbot e sfruttare le funzionalità dell'applicazione.

[2]modulo che comprende funzioni per aprire URL in applicazioni che interagiscono con un browser

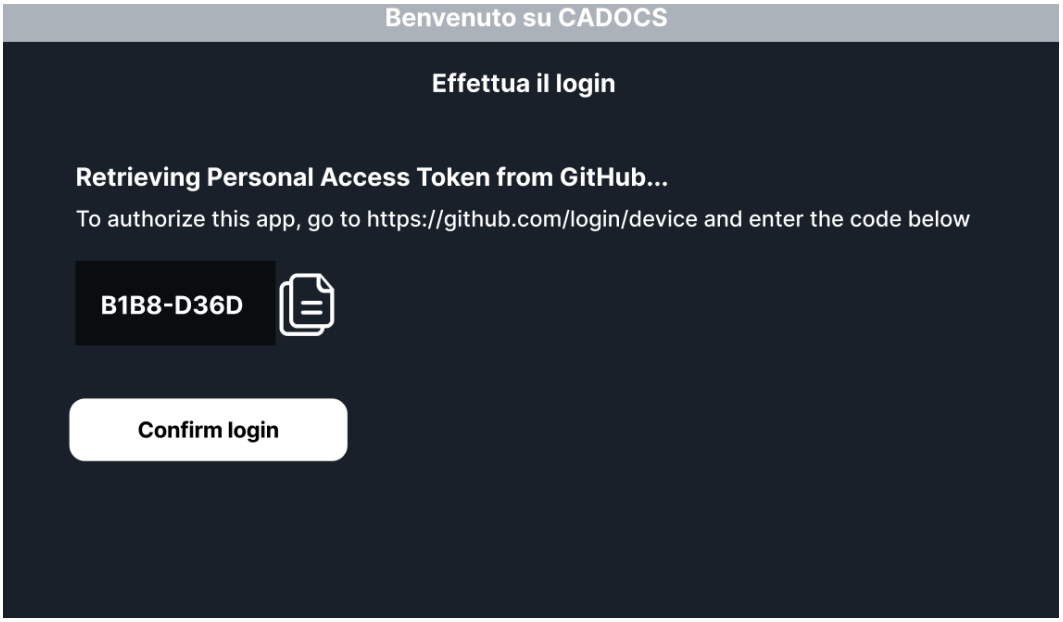

Figura 4.3: GUI login dopo aver cliccato il pulsante

4.3.2 Pagina Home

Dopo aver completato il login, l'utente viene automaticamente reindirizzato alla pagina principale dell'applicazione (facendo riferimento alla Figura 4.4). Qui è presente un'area *EntryWidget*, che consente all'utente di inserire il proprio messaggio. Una volta digitato il messaggio, questo viene elaborato dalla classe *IntentManager*,viene inviato al modello NLU (Natural Language Understanding) per l'analisi. Il modello elabora prima di tutto il linguaggio con la quale l'utente sta parlando al chatbot, poi ne estrae l'intent, inviando il risultato della classe IntentManager alla classe *IntentResolver*. Sulla base del risultato ottenuto dalla classe intentResolver, viene generata una risposta appropriata alla domanda posta dall'utente, offrendo un'interazione significativa e una corretta risposta alle richieste.

Figura 4.4: Home di CADOCS

Il chatbot è progettato per rilevare quattro tipi di intent:

- **GetSmells**, quando rilevato quest'intent, viene richiamato il webService di csDetector per la detection dei community smell della *entity* rilevata nel messaggio inviato. Un esempio di messaggio può essere: *hello CADOCS, show me the community smells in the repository https://github.com/tensorflow/ranking*, in questo caso l'unica entity del messaggio è il link della repository.

- **GetSmellsDate**, quando rilevato quest'intent, viene effettuata la stessa operazione del GetSmells ma la detection avviene sulla base della data inviata nel messaggio. Un esempio di messaggio può essere: *hello CADOCS, show me the community smells in the repository https://github.com/tensorflow/ranking from 21/05/2020*, in questo caso ci sono due entity, il link della repository e la data da dove iniziare la detection.

- **Report**, quando rilevato quest'intent, viene inviato un messaggio contenente il report dell'ultima esecuzione del tool.

- **Info**, quando rilevato quest'intent, viene inviato un messaggio contenente le informazioni sui community smells, accompagnati da link utili per contattare gli sviluppatori.

Inoltre, quando il modello NLU non rileva nessun intent nel nostro messaggio, viene stampato un messaggio di errore, per avvisare l'utente che, probabilmente, avrà inviato un messaggio sbagliato.

4.3.3 Gestione customException

Tra le motivazioni principali alla base della creazione di questo tool c'è anche la gestione delle eccezioni personalizzate, che come specificato nel capitolo precedente, sono state implementate in csDetector nel caso di errori trovati nel lavoro di manutenzione effettuato sullo strumento.

In csDetector, l'avvio della rilevazione dei community smells su una repository può generare un errore, quindi sollevare un'eccezione specifica, identificata da un codice personalizzato. Questo tool si occupa di controllare attentamente la risposta ottenuta dal webService, verificando la presenza di questo codice personalizzato, l'obiettivo è catturare l'eccezione corrispondente e fornire informazioni esaustive all'utente. Questo processo mira a spiegare chiaramente la causa del malfunzionamento del tool, consentendo all'utente di comprendere la situazione e i motivi per cui l'esecuzione è stata interrotta. La Figura sottostante mostra come viene catturata e gestita l'eccezione [4.5]

```
if req.status_code == 890:
    error_text = response_json.get('error')
    code = response_json.get('code')
    results = [error_text, code]
    return results
```

Figura 4.5: Eccezione catturata dalla risposta del webService

4.3.4 Creazione del file eseguibile

Una volta ultimata l'implementazione del tool siamo passati alla creazione del file eseguibile tramite la libreria Pyinstaller. Inizialmente, è stata effettuata un'analisi delle dipendenze del tool sviluppato, incluse le librerie Python utilizzate e i moduli esterni necessari per il suo corretto funzionamento.

La fase successiva ha riguardato la creazione effettiva dell'installer. Pyinstaller è stato utilizzato per creare un pacchetto eseguibile del tool, assicurandosi che tutte le risorse e le dipendenze fossero correttamente integrate nell'installer, in modo che il tool potesse essere eseguito senza dover preoccuparsi delle dipendenze esterne. Tramite questo comando mostrato in basso, viene creato l'installer con tutte le dipendenze necessarie per il suo funzionamento:

Comando per creare l'installer

pyinstaller toolGui.py –onefile –copy-metadata tqdm –copy-metadata regex –copy-metadata requests –copy-metadata packaging –copy-metadata filelock –copy-metadata numpy –copy-metadata tokenizers –copy-metadata huggingface-hub –copy-metadata safetensors –copy-metadata pyyaml –copy-metadata torch –add-data src/.env:.

Una volta avviato il comando tramite linea di comando, Pyinstaller mi genera due *directory*, ed un file:

- **Directory build**, contenente tutti i file utili all'installer per avviarsi correttamente.
- **Directory dist**, contenente il file **.exe* vero e proprio.
- **File *.spec**, si tratta di un file di specifica del nostro tool, modificabile e in cui è necessario elencare tutte le dipendenze e i moduli aggiuntivi essenziali per il corretto funzionamento del programma.

CAPITOLO 5

Conclusioni

In quest'ultimo capitolo, andremo a riflettere su quanto fatto finora, sulle difficoltà affrontate durante lo sviluppo e, soprattutto, sui possibili sviluppi futuri del nostro software. Dovremo quindi inquadrare quelle che sono le criticità del lavoro svolto, per poi riflettere su come sarebbe possibile risolverle in una eventuale futura versione.

Ora che lo sviluppo dell'installer e del tool è giunto al termine, possiamo effettuare una valutazione sul lavoro svolto, confrontando il risultato con le aspettative iniziali.

Lo scopo iniziale di questa tesi era focalizzato sulla manutenzione e sull'evoluzione dei sistemi csDetector e CADOCS. L'intento era migliorare la versatilità e la facilità d'uso di entrambi. Gli obiettivi iniziali di questo lavoro erano i seguenti:

- Migliorare la gestione degli errori di csDetector.
- Integrare gli altri lavori sia in csDetector che in CADOCS.
- Trasformare CADOCS in una desktop application, in modo tale da renderla accessibile a un pubblico più ampio, consentendone l'utilizzo da parte di un numero maggiore di utenti.

La fase iniziale del mio lavoro si è concentrata sulla manutenzione di csDetector, lavoro che si suddivideva in: gestione degli errori e l'unione con CADOCS.

Posso dire che, per quanto riguarda la gestione degli errori, gli obiettivi prefissati inizialemente sono tutti stati raggiunti con un buon risultato.

Invece, l'idea iniziale di unire il webService di csDetector con il chatbot di CADOCS in un'unica applicazione è stata accantonata a causa di problemi di compatibilità tra diverse versioni di Python dei due sistemi. Questo avrebbe richiesto un processo complesso e non c'erano idee chiare per eseguire un merge efficace. Di conseguenza, la scelta è stata quella di concentrarsi sulla creazione di un installer, rendendo l'installazione più semplice e accessibile a una gamma più ampia di utenti.

Per quanto riguarda CADOCS, lo sviluppo del tool ha prodotto un sistema che risponde efficacemente con quanto dichiarato nei requisiti funzionali. Il sistema prevede quindi le funzionalità necessarie previste inizialmente come:

- Detection dei community smells totali in una repository.
- Detection dei community smells successivi ad una determinata data di inizio.
- Un sistema di login tramite GitHub.
- Integrazione del modello NLU.

Non siamo riusciti a soddisfare tutti i requisiti non funzionali che ci eravamo prefissati. L'interfaccia non rispetta pienamente gli standard di qualità che volevamo raggiungere. Attualmente CADOCS si limita a mostrare a schermo la risposta ricevuta da csDetector, quando sarebbe stato preferibile avere un'interfaccia più curata e con qualche ulteriore funzionalità rispetto a quelle proposte.

L'idea iniziale prevedeva inoltre, l'utilizzo di pulsanti dedicati, ognuno associato a un *intent* specifico. Cliccando su un pulsante, si attivava l'intent corrispondente, richiedendo le *entity* necessarie per l'esecuzione, avviandolo automaticamente.

5.1 Sviluppi futuri

Di seguito si andranno ad elencare i possibili sviluppi futuri individuati nel contesto del presente lavoro di tesi.

5.1.1 csDetector

Relativamente al lavoro su csDetector, uno degli aspetti fondamentali riguarda il miglioramento delle situazioni in cui csDetector interrompe il proprio processo. Quindi l'implementazione di utleriori nuove soluzioni per prevenire tali problemi.

Per l'installer invece, oltre qualche miglioramento generale di bug, ed implementazione di ulteriori controlli, le sistemazioni principali da fare sono due:

- Aumentare la portabilità anche su altri sistemi. Permettere l'utilizzo del tool sulla più grande quantità di macchine possibili.
- Cercare di automatizzare l'inserimento automatico nella cartella *SentiStrenght* dei file necessari per le sue operazioni.

5.1.2 CADOCS

Relativamente al lavoro sull'evoluzione di CADOCS. Tra le principali migliorie da considerare per sviluppi futuri del progetto, ci sono le seguenti:

- Come anticipato in precedenza, bisogna sicuramente migliorare notevolmente l'interfaccia utente, miglioramenti grafici e implementazione di nuove funzionalità. Un obiettivo chiave è l'aggiunta di pulsanti dedicati a ciascun intent, per ottimizzare anche le prestazioni.
- La possibile implementazione di un nuovo intent, denominato *GetSmellsBeginEndDate*, consentirebbe la rilevazione di smells, specificando un intervallo di date di inizio e fine per l'analisi.
- ottimizzazione dell'eseguibile generato da PyInstaller. Cioè renderlo compatibile con diverse piattaforme. Oltre che a risolvere i bug che attualmente emergono durante l'utilizzo dell'applicazione.

La repository sia del nuovo csDetector[1], sia di CADOCS 2.0[2] con tutte le indicazioni per usarli, sono disponibili in due repository online.

[1]Link alla repository di csDetector: [https://github.com/PaoloCarmine1201/csDetector]
[2]Link alla repository di CADOCS 2.0: [https://github.com/PaoloCarmine1201/CADOCS_II]

Bibliografia

[1] G. Voria, V. Pentangelo, A. D. Porta, S. Lambiase, G. Catolino, F. Palomba, and F. Ferrucci, "Community smell detection and refactoring in slack: The cadocs project," *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022. (Citato alle pagine iii e 11)

[2] I. Maglogiannis, L. Iliadis, and E. Pimenidis, "Scopus - document search," *Springer Nature - PMC COVID-19 Collection*, 2020. (Citato alle pagine iii e 14)

[3] D. A. Tamburri and E. Di Nitto, "When software architecture leads to social debt," *2015 12th Working IEEE/IFIP Conference on Software Architecture*, 2015. (Citato a pagina 7)

[4] G. Catolino, F. Palomba, D. A. Tamburri, A. Serebrenik, and F. Ferrucci, "Gender diversity and women in software teams: How do they affect community smells?" *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 2019. (Citato a pagina 7)

[5] F. Palomba, D. Andrew Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?" *IEEE Transactions on Software Engineering*, 2021. (Citato a pagina 7)

[6] D. A. Tamburri, R. Kazman, and H. Fahimi, "The architect's role in community shepherding," *IEEE Software*, 2016. (Citato a pagina 7)

[7] G. Catolino, F. Palomba, D. A. Tamburri, A. Serebrenik, and F. Ferrucci, "Gender diversity and community smells: Insights from the trenches," *IEEE Software*, 2020. (Citato a pagina 7)

[8] Tamburri and D. A., "Software architecture social debt: Managing the incommunicability factor," *IEEE Transactions on Computational Social Systems*, 2019. (Citato a pagina 7)

[9] D. A. Tamburri, P. Kruchten, P. Lago, and H. van Vliet, "Social debt in software engineering: insights from industry," *Journal of Internet Services and Applications*, 2015. (Citato a pagina 7)

[10] D. A. Tamburri, F. Palomba, and R. Kazman, "Exploring community smells in open-source: An automated approach," *IEEE Transactions on Software Engineering*, 2021. (Citato a pagina 7)

[11] M. C. I. S. Nuri Almarimi, Ali Ouni and M. W. Mkaouer, "Learning to detect community smells in open source software projects," *Proceedings of the 15th International Conference on Global Software Engineering*, 2020. (Citato a pagina 9)

[12] D. A. T. Fabio Palomba, "Predicting the emergence of community smells using socio-technical metrics: A machine-learning approach," *Journal of Systems and Software*, 2020. (Citato a pagina 9)

[13] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle, "From developer networks to verified communities: A fine-grained approach," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015. (Citato a pagina 9)

[14] N. Almarimi, A. Ouni, M. Chouchen, and M. W. Mkaouer, "Csdetector: An open source tool for community smells detection," 2021. (Citato alle pagine 9 e 10)

[15] B. A. Shawar and E. Atwell, "Chatbots: are they really useful?" *Journal for Language Technology and Computational Linguistics*, 2007. (Citato a pagina 12)

[16] A. Chopra, A. Prashar, and C. Sain, "Natural language processing," *International journal of technology enhancements and emerging engineering research*, 2013. (Citato a pagina 12)

[17] H.-Y. Shum, X.-d. He, and D. Li, "From eliza to xiaoice: challenges and opportunities with social chatbots," *Frontiers of Information Technology & Electronic Engineering*, 2018. (Citato a pagina 13)

[18] E. Adamopoulou and L. Moussiades, "An overview of chatbot technology," *IFIP international conference on artificial intelligence applications and innovations*, 2020. (Citato a pagina 13)

[19] P. Costa, "Conversing with personal digital assistants: On gender and artificial intelligence," *Journal of Science and Technology of the Arts*, 2018. (Citato alle pagine 14 e 16)

[20] P. B. Brandtzaeg and A. Følstad, "Why people use chatbots," *Internet Science: 4th International Conference, INSCI 2017, Thessaloniki, Greece, November 22-24, 2017, Proceedings 4*, 2017. (Citato a pagina 14)

[21] K. Ramesh, S. Ravishankaran, A. Joshi, and K. Chandrasekaran, "A survey of design techniques for conversational agents," *International conference on information, communication and computing technology*, 2017. (Citato a pagina 15)

[22] ——, "A survey of design techniques for conversational agents," 2017. (Citato a pagina 18)

[23] Y. Wu, W. Wu, C. Xing, M. Zhou, and Z. Li, "Sequential matching network: A new architecture for multi-turn response selection in retrieval-based chatbots," 2017. (Citato a pagina 18)

[24] J. P. Chang, C. Chiam, L. Fu, A. Z. Wang, J. Zhang, and C. Danescu-Niculescu-Mizil, "Convokit: A toolkit for the analysis of conversations," *arXiv preprint arXiv:2005.04246*, 2020. (Citato a pagina 24)