UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Progettazione e realizzazione di un chatbot a supporto delle attività del Sindacato Italiano Militari Carabinieri

Design and implementation of a chatbot to support the activities of "Sindacato Italiano Militari Carabinieri"

Relatore

Prof. Domenico Ursino

Candidato

Donato Di Zinno

Corelatore

Prof. Francesco Cauteruccio

ANNO ACCADEMICO 2022-2023

“Il giorno più bello? Oggi.
L’ostacolo più grande? La paura.
La cosa più facile? Sbagliarsi.
L’errore più grande? Rinunciare.
La felicità più grande? Essere utili agli altri.
Il sentimento più brutto? Il rancore.
Il regalo più bello? Il perdono.
Quello indispensabile? La famiglia.”

Madre Teresa di Calcutta, "Il Giorno Più bello"’

Sommario

Il lavoro che è stato svolto si basa sull'analisi e sulla progettazione e implementazione di un chatbot utilizzando il Natural Language Processing (NLP) e il framework Rasa. Lo studio si concentra sull'integrazione del chatbot nel sistema web del Sindacato Italiano Militari dei Carabinieri.

Nel corso del lavoro, vengono esaminate le motivazioni che spingono le aziende ad adottare un chatbot e vengono illustrate le componenti e gli strumenti software necessari per la creazione di un assistente virtuale. L'NLP viene utilizzato per migliorare la precisione e l'efficienza del chatbot.

Successivamente, viene eseguita l'implementazione del chatbot nel sistema web e nell'app Telegram. Al fine di facilitare il suo utilizzo, viene redatto un manuale utente specifico che fornisce istruzioni dettagliate sull'implementazione e l'utilizzo del chatbot.

In sintesi, questo studio fornisce una panoramica completa sulla progettazione, programmazione e implementazione di un chatbot utilizzando l'NLP e il framework Rasa.

Keyword: Chatbot, Natural Language Processing, Sindacato Italiano Militari dei Carabinieri, Rasa, Intelligenza Artificiale.

Indice

Introduzione **1**

1 Introduzione ai chatbot **3**
1.1 Funzionalità dei chatbot 3
1.2 L'intelligenza artificiale nei Chatbot 4
1.3 Vantaggi e limiti dei Chatbot 6
1.3.1 I vantaggi 6
1.3.2 I Limiti 7

2 Strumenti software utilizzati **8**
2.1 Ambiente di lavoro 8
2.1.1 Anaconda 9
2.2 Rasa 10
2.2.1 Rasa Framework 10
2.2.2 Funzionamento di Rasa 10
2.2.3 Implementazione con altri sistemi 13
2.3 Librerie utilizzate 14

3 Specifica e analisi dei requisiti **17**
3.1 Contesto di utilizzo 17
3.1.1 Sezione pubblica 18
3.1.2 Sezione privata 20
3.2 Analisi dei requisiti 21

4 Progettazione **22**
4.1 Installazione e Configurazione 22
4.1.1 Installazione della macchina virtuale 23
4.1.2 Installazione di Rasa 28
4.2 Scelta del modello NLP 30
4.3 Progettazione chatbot 31
4.3.1 Progettazione degli intenti 34
4.3.2 Progettazione delle storie 36
4.3.3 Progettazione delle regole 38
4.4 Progettazione delle risposte e delle azioni del chatbot 39
4.4.1 Fallback Action 45

4.4.2 Realizzazione della funzione "ricerca segretario" . . . 46
4.4.3 Jaccard e Set-Containment . . . 49
4.4.4 Realizzazione della funzione "ricerca allegato" . . . 50

5 Implementazione **55**
5.1 Inizializzazione del chatbot . . . 55
5.2 Implementazione web del chatbot . . . 62
5.3 Implementazione del chatbot su Telegram . . . 64

6 Manuale Utente **68**
6.1 Introduzione . . . 68
6.2 Navigazione dell'Interfaccia . . . 68
6.3 Utilizzo delle funzionalità . . . 71
6.3.1 Funzionalità relative alla Sezione Pubblica . . . 71
6.3.2 Funzionalità Sezione Privata . . . 75
6.3.3 Gestione degli errori . . . 78

7 Discussione in merito al lavoro svolto **81**
7.1 Contestualizzazione del lavoro . . . 81
7.2 Analisi dei risultati . . . 81
7.3 Valutazione delle prestazioni . . . 82

8 Conclusioni e uno sguardo al futuro **83**

Bibliografia **84**

Ringraziamenti **86**

Elenco delle figure

1.1 Fasi dell'Intelligenza Artificiale 5
1.2 Punti di Eccellenza e della macchina 5

2.1 Visual Studio Code 9
2.2 Anaconda 9
2.3 Rasa 10
2.4 Architettura di Rasa 10
2.5 Principali Piattaforme con cui si può integrare Rasa 13
2.6 Rasa Sdk 14
2.7 Pandas 14
2.8 TensorFlow 15
2.9 Datasketch 15

3.1 Sindacato Italiano Militari Carabinieri 17
3.2 Fermo Tech 18

4.1 Architettura M1 (ARM) 22
4.2 Il logo di UTM 24
4.3 UTM: Crezione macchina virtuale 24
4.4 UTM: Scelta della tipologia di virtualizzazione 25
4.5 UTM: Scelta del sistema operativo da virtualizzare 25
4.6 UTM: Configurazione dell'avvio dell'installazione di Windows 10 26
4.7 UTM: Configurazione della quantità di RAM e dei processori per Windows 10 26
4.8 UTM: Configurazione della quantità di Hard Disk da allocare a Windows 10 . 27
4.9 UTM: Scelte opzionali e conclusione della configurazione 27
4.10 UTM: Macchina Virtuale 28
4.11 UTM: Configurazioni personalizzate 28
4.12 Prompt dei comandi di Anaconda 29
4.13 Comando per la creazione della variabile *env* 29
4.14 Comando per l'attivazione della variabile *env* 29
4.15 Verifica dell'attivazione della variabile *env* 30
4.16 Pipeline di Rasa 32
4.17 Policy Rasa 33
4.18 Intento saluto 34
4.19 Intenti creati 34

4.20 Intenti creati . . . 35
4.21 Intento Cerca Segretario Regione . . . 35
4.22 Lookup Table . . . 36
4.23 Esempio di una storia . . . 36
4.24 Utilizzo del checkpoint . . . 37
4.25 Storie . . . 37
4.26 Esempio di una regola . . . 38
4.27 Condizione *conversation_start* . . . 38
4.28 Condizione *slot_was_set* . . . 38
4.29 Regole . . . 39
4.30 Intenti, Entity, e Action del file *domain* . . . 40
4.31 Mappatura degli slot e delle entity . . . 40
4.32 Risposte del Chatbot . . . 41
4.33 Risposte del chatbot con pulsanti interattivi . . . 41
4.34 Utter relativo all'iscrizione al sito . . . 42
4.35 Utter relativo al reset della password . . . 42
4.36 Utter relativo al trattamento previdenziale . . . 42
4.37 Utter relativo al trattamento stipendiale . . . 42
4.38 Utter relativo alla scelta del trattamento . . . 43
4.39 *SecretaryInizioAction* . . . 44
4.40 *SecretaryPolizzaAction-1* . . . 44
4.41 *SecretaryPolizzaAction-2* . . . 45
4.42 Core *FallBack-config.yml* . . . 45
4.43 Fallback *Classifier-config.yml* . . . 45
4.44 Action *Fallback-1* . . . 46
4.45 Action *Fallback-2* . . . 46
4.46 Action *Cerca Segretario di Regione-1* . . . 47
4.47 Action *Cerca Segretario di Regione-2* . . . 47
4.48 Action per la scelta della provincia . . . 48
4.49 Action Cerca Provincia . . . 48
4.50 Action per la gestione della ricerca del segretario di regione . . . 49
4.51 Jaccard e Set-Containment . . . 50
4.52 Azione per la Ricerca dell'allegato . . . 51
4.53 Funzione *find_word_in_file* . . . 51
4.54 SequenceMatcher . . . 52
4.55 *LSHEnsamble* . . . 53
4.56 Funzione per il calcolo del coefficente di Jaccard . . . 53
4.57 Funzione per ottenere i link degli allegati e il file txt . . . 54

5.1 Posizione del percorso del chatbot . . . 55
5.2 Numero di epoche-*config.yml* . . . 56
5.3 Crezione di un modello in Rasa . . . 56
5.4 Nuova shell per l'avvio del server delle azioni personalizzate . . . 57
5.5 Comando *rasa run actions* . . . 57
5.6 Comando *Server per le azioni personalizzate* . . . 57
5.7 Istruzione per il funzionamento del server delle action . . . 57
5.8 Funzionamento di *rasa_debug* . . . 58
5.9 Funzionamento di *rasa_interactive*-1 . . . 59
5.10 Funzionamento di *rasa_interactive*-2 . . . 60
5.11 Esempio di un diagramma di flusso di una conversazione . . . 61
5.12 Link per il diagramma di flusso . . . 61

5.13 File *index.html* 64
5.14 Procedura di registrazione del chatbot su Telegram-1 64
5.15 Procedura di registrazione del chatbot su Telegram-2 65
5.16 Procedura di registrazione del chatbot su Telegram-3 65
5.17 Attivazione del servizio per l'app di messaggistica Telegram 66
5.18 Server rasa funzionante 67

6.1 Interfaccia del chatbot in un sito internet 69
6.2 Finestra della chat 70
6.3 Saluto e lista delle funzionalità del chatbot (parte pubblica e parte privata) . . 70
6.4 Risposta con pulsanti 71
6.5 Risposta con immagini 71
6.6 Procedura per l'iscrizione al sito 72
6.7 Procedura per il recupero della password 72
6.8 Modalità per l'attivazione della procedura della ricerca del Segretario di regione 73
6.9 Azione per cercare il segretario di provincia 74
6.10 Messaggio di *Arrivederci* 74
6.11 Terza modalità per l'attivazione della funzione ricerca segretario di regione . 75
6.12 Procedura per il trattamento pensionistico 76
6.13 Procedura per il trattamento stipendiale 76
6.14 Attivazione della funzione di ricerca dell'allegato 77
6.15 Istruzione per fermare la funzione di ricerca dell'allegato 77
6.16 Procedura per la gestione della polizza e dei sinistri 78
6.17 Gestione del mancato riconoscimento dell'intenzione 79
6.18 Gestione della funzione: ricerca Segretario di Regione 79
6.19 Gestione della funzione relativa alla gestione del trattamento Pensionistico e Stipendiale 80
6.20 Gestione della funzione relativa alla ricerca dell'allegato 80

Introduzione

Negli ultimi anni, l'Intelligenza Artificiale (IA) ha rivoluzionato molti settori e le aziende stanno sempre più considerando l'adozione di soluzioni innovative per migliorare l'interazione con i propri clienti. In questo contesto, i chatbot hanno guadagnato un ruolo di rilievo come assistenti virtuali capaci di offrire un'esperienza personalizzata e immediata agli utenti.

L'introduzione di un chatbot all'interno di un'azienda può offrire numerosi vantaggi. Innanzitutto, la disponibilità costante di un assistente virtuale permette di fornire supporto continuo ai clienti, migliorando la loro soddisfazione e fidelizzazione. Inoltre, i chatbot possono automatizzare compiti ripetitivi e di routine, riducendo gli errori umani e gestendo un alto volume di interazioni senza la necessità di un supporto umano costante.

Grazie all'IA e all'elaborazione del linguaggio naturale (Natural Language Processing NLP), i chatbot possono scalare facilmente per gestire un numero crescente di interazioni con gli utenti. Per garantire un'interazione naturale ed efficace con gli utenti, i chatbot si basano sull'NLP. L'NLP permette ai chatbot di comprendere e interpretare il linguaggio umano, consentendo loro di rispondere in modo coerente e contestuale alle richieste degli utenti.

Ciò permette alle aziende di soddisfare la domanda in modo rapido ed efficiente, riducendo i tempi di attesa, senza dover investire risorse significative nella formazione e nell'espansione del personale. A questo proposito, tali tecnologie consentono alle aziende di ripartire i dipendenti in altre attività strategiche e permettono ad esse di ridurre i costi operativi e di ottenere un aumento della produttività complessiva.

Con l'IA e l'NLP le aziende possono analizzare grandi quantità di dati non strutturati, come testi o conversazioni basate sul testo, ed estrarre informazioni rilevanti. Ciò consente di ottenere una migliore comprensione delle esigenze dei clienti, di identificare tendenze di mercato e prendere decisioni informate basate su dati.

L'integrazione dell'IA e dell'NLP all'interno di un chatbot rappresentano, quindi, degli elementi cruciali per garantire un'esperienza utente di qualità. Il loro utilizzo consente ai chatbot di apprendere e migliorare nel tempo, adattandosi alle esigenze specifiche dell'azienda. A tal riguardo, un chatbot può essere implementato su diverse piattaforme, come il sito web dell'azienda, le app di messaggistica o i social media. Ciò permette di offrire un supporto coerente e integrato su più canali.

Negli ultimi anni, le aziende si sono impegnate nella ricerca del progresso tecnologico, abbracciando l'implementazione di un chatbot come parte integrante del loro processo produttivo. Questo significativo passo verso l'innovazione tecnologica dimostra la volontà dell'azienda di essere all'avanguardia, adattandosi alle nuove tendenze e rispondendo alle esigenze dei clienti in modo sempre più efficace e tempestivo.

La presente tesi si colloca in tale contesto applicativo ; infatti, essa illustrerà la progettazione, l'implementazione e l'integrazione di un chatbot su diverse piattaforme. La prima parte del lavoro si concentrerà sugli studi e sulle analisi delle motivazioni che spingono le aziende ad adottare un chatbot nei loro sistemi. In particolare, verranno approfondite le funzionalità dei chatbot, l'Intelligenza Artificiale e la sua implementazione in essi, cercando di valutarne vantaggi e limiti potenziali come soluzione.

Successivamente, saranno presentati gli strumenti software utilizzati e le varie componenti necessarie per la creazione dell'assistente virtuale. Si sottolineerà l'importanza fondamentale di ciascun elemento, poiché la mancanza di uno di essi potrebbe compromettere la riuscita del progetto.

Il nucleo centrale della tesi sarà dedicato all'analisi dettagliata della progettazione del sistema e della logica che costituisce il chatbot. Saranno affrontati gli aspetti e le problematiche da considerare per garantire il successo nella sua creazione, tenendo conto delle richieste commissionate. Saranno, inoltre, utilizzati algoritmi di NLP e saranno sviluppati appositi algoritmi per ottimizzare le funzioni del chatbot. Verranno condotte ricerche e approfondimenti sull'uso del coefficiente di Jaccard e del set-containment, cercando di trovare il giusto compromesso tra efficienza e accuratezza.

Dopo la fase di progettazione, si procederà con l'implementazione del chatbot sia su un sistema web che su una delle app di messaggistica più popolari, come Telegram, fornendo tutti i comandi e le istruzioni necessarie per agevolare gli utilizzatori e consentire un utilizzo completo. A tal fine, sarà redatto un manuale utente specifico per l'implementazione del chatbot su un sistema web.

La tesi si concluderà con una discussione approfondita sul lavoro svolto, valutando la sua contestualizzazione, analizzando obiettivamente i risultati ottenuti e valutandone le prestazioni in termini di accuratezza, efficienza e tempi di risposta.

La presente tesi è strutturata come di seguito specificato:

- Nel Capitolo 1 verrà introdotto l'ambito dell'Intelligenza Artificiale e dei chatbot.
- Nel Capitolo 2 si descriveranno gli strumenti hardware e i software utilizzati.
- Nel Capitolo 3 si descrivernno le richieste e il processo di analisi dei requisiti.
- Nel Capitolo 4 si illustreranno il processo di progettazione e la logica del chatbot.
- Nel Capitolo 5 verranno descritte le modalità di implementazione per un sistema web e per un sistema di app di messaggistica.
- Nel Capitolo 6 sarà redatto un manuale utente che fornirà indicazioni all'utente su come utilizzare il chatbot sviluppato.
- Nel Capitolo 7 verranno espresse delle considerazioni riguardo al lavoro svolto.
- Nel Capitolo 8 saranno tratte le conclusioni, mostrando dei possibili approfondimenti futuri.

CAPITOLO 1

Introduzione ai chatbot

In questo capitolo verrà introdotto il concetto di chatbot e come si è evoluta la sua funzione nel tempo. La sua prima apparizione avvenne negli anni '60 con l'obbiettivo di simulare una conversazione terapeutica. Questo primo prototipo era una semplice interfaccia basata principalmente sul testo con l'intento di analizzare le risposte date dall'utenza così da generare una risposta adeguata.Negli anni'70 con l'avvento dell'intelligenza artificiale, questa tecnologia approdò nel mondo videoludico con l'obbiettivo di creare un'interazione con gli utenti nell'ambiente di gioco. Con la nascita di internet e del web, negli anni '90 i chatbot iniziano a diffondersi con lo scopo di prestare assistenza ai clienti.Infine, negli ultimi anni, con l'affermarsi delle appicazioni di messagistica e dei social media, l'uso del chatbot è aumentato, sopratutto per la gestione delle conversazioni e dei servizi automatizzati.I chatbot ad oggi vengono utilizzati in molteplici campi.

1.1 Funzionalità dei chatbot

Un chatbot è un programma informatico progettato per simulare una conversazione umana con gli utenti attraverso l'uso di chat o messaggi di testo. I chatbot, diventati ormai sempre più popolari, grazie alle loro molteplici funzionalità, hanno permesso di trarne beneficio a tutti coloro che ne usufruiscono.

Esistono due principali tipi di chatbot:

- I chatbot dedicati alle attività (dichiarativi): sono programmi utilizzati per l'esecuzione di una funzione. Essi fanno uso di regole NLP ("Natural Language Process") e pochissima ML ("Machine Learning") e generano risposte automatizzate. Le interazioni si limitano solamente a delle specifiche ben precise e strutturate. Sebbene utilizzino l'NLP in modo tale che gli utenti finali possano sperimentarli in modo semplice, le loro capacità sono abbastanza basilari. Attualmente, questi sono i chatbot più usati.

- I chatbot predittivi basati sui dati (di conversazione): sono molto più sofisticati, interattivi e personalizzati rispetto ai chatbot dichiarativi. Questi chatbot riescono a rendersi conto del contesto di riferimento e cercano, attraverso la comprensione del linguaggio naturale, di imparare applicando l'intelligenza predittiva e l'analisi dei dati per consentire la personalizzazione degli utenti. Gli assistenti digitali impiegano tempo per apprendere. Infine, oltre a monitorare i dati e a seguire delle linee guida, essi possono anche avviare delle conversazioni. Siri di Apple e Alexa di Amazon sono esempi di chatbot predittivi orientati al consumatore e basati sui dati.

Gli assistenti digitali molto avanzati hanno anche la capacità di connettere diversi chatbot monouso sotto un unico gruppo, di estrarre diverse informazioni da ciascuno di essi e

di combinare le informazioni ricevute per eseguire un'attività mantenendo, comunque, il contesto.

Le funzionalità principali per cui vengono utilizzati i chatbot sono:

- assistenza clienti
- automazione
- marketing
- ricerca e Analisi.

L'assistenza clienti è una funzionalità basata su un sistema di messaggistica con l'interazione di un operatore o di un automa per poter gestire le eventuali richieste di un utente.

L'automazione è una caratteristica che permette di automatizzare determinate attività ripetitive, in modo da poter ridurre i costi operativi.

Per quanto concerne il marketing, invece, i chatbot possono essere utili per promuovere prodotti e servizi. Infine, l'utilizzo per la ricerca e analisi da la possibilità di raccogliere dati con lo scopo di analizzare le tendenze di mercato e di migliorare strategie di business.

Benchè quelle appena elencate siano le principali, le potenzialità dei chatbot sono maggiori. Con il passare del tempo e con i progressi nel campo dell'intelligenza artificiale, la loro applicazione diventa sempre più diffusa.

1.2 L'intelligenza artificiale nei Chatbot

Il termine "intelligenza artificiale" (IA) è nato negli anni '50 durante la conferenza presso il Dartmouth College ad opera del matematico John McCarthy. I fondamenti dell'*intelligenza artificiale* risalgono al XV secolo con i lavori pioneristici di Leonardo da Vinci, come il "Pensatore Meccanico", e ,successivamente, con la macchina analitica di Charles Babbage nel 1837.

Oggetto di interesse e di studio per molti studiosi e ricercatori, le prime difficoltà riscontrate riguardavano la totale mancanza di conoscenza semantica relativa ai domini trattati dalle macchine. La loro capacità di ragionamento si limitava, infatti, a una semplice manipolazione sintattica.

A metà degli anni '80, l'introduzione delle reti neurali ha permesso di utilizzare algoritmi di apprendimento in grado di fornire un supporto alla semantica.

L'integrazione della IA nei chatbot ha permesso di apprendere dai dati ricevuti in input e di migliorare le loro prestazioni nel tempo. Una possibile applicazione dell'intelligenza artificiale odierna è il suo utilizzo in macchinari capaci di replicare le azioni umane. Il concetto di intelligenza artificiale può essere, quindi, concepito come una simbiosi tra uomo e macchina. In questa prospettiva la macchina non imita l'uomo, bensì viene a crearsi un ecosistema uomo-macchina in cui sia l'uomo che la macchina mettono ciascuno il meglio di sé; cio che ne deriva migliora le capacità di entrambe le componenti dell'ecosistema.

Nell'informatica si possono distinguere tre ere (Figura 1.1):

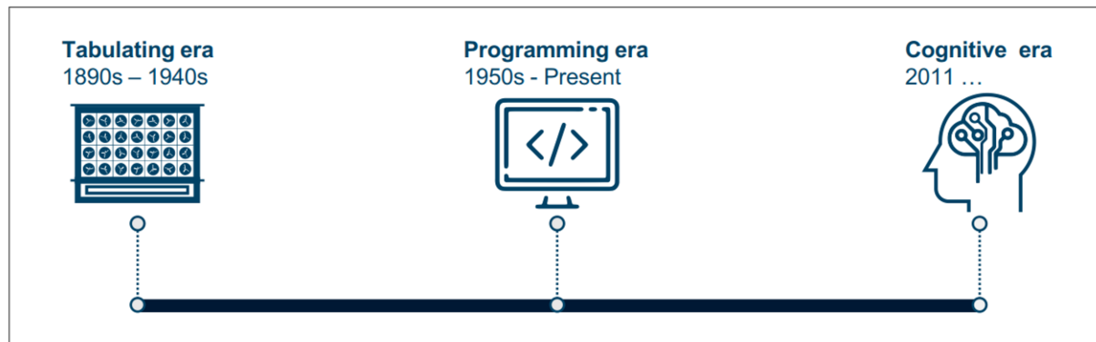

Figura 1.1: Fasi dell'Intelligenza Artificiale

In particolare:

1. Durante la prima era, ovvero prima della nascita della macchina di Turing, ogni macchina effettuava un unico compito (si pensi, ad esempio, alle calcolatrici).

2. La seconda era è quella della programmazione. Quest'ultima rappresentava il primo tentativo di fare in modo che una macchina non sia specifica per un compito, ma il compito della macchina cambi a seconda del programma che sta girando su di essa. Quest'era inizia negli anni '50 con Turing e sta continuando ancora oggi.

3. La terza era, che affiancherà la seconda, è quella del Cognitive Computing. Essa parte circa dal 2011.

L'idea del Cognitive Computing è di non simulare l'uomo, ma creare una forte interazione, fra computer ed esso. Il computer dalla sua ha la velocità e la capacità di calcolo, di conoscenza, etc. mentre l'uomo ha tutta una serie di capacità che lo caratterizzano rispetto al computer. Tutto ciò vien evidenziato nella Figura 1.2.(ragionamento, astrazione, intuito, avere un'etica e una morale, essere pragmatico, etc.).

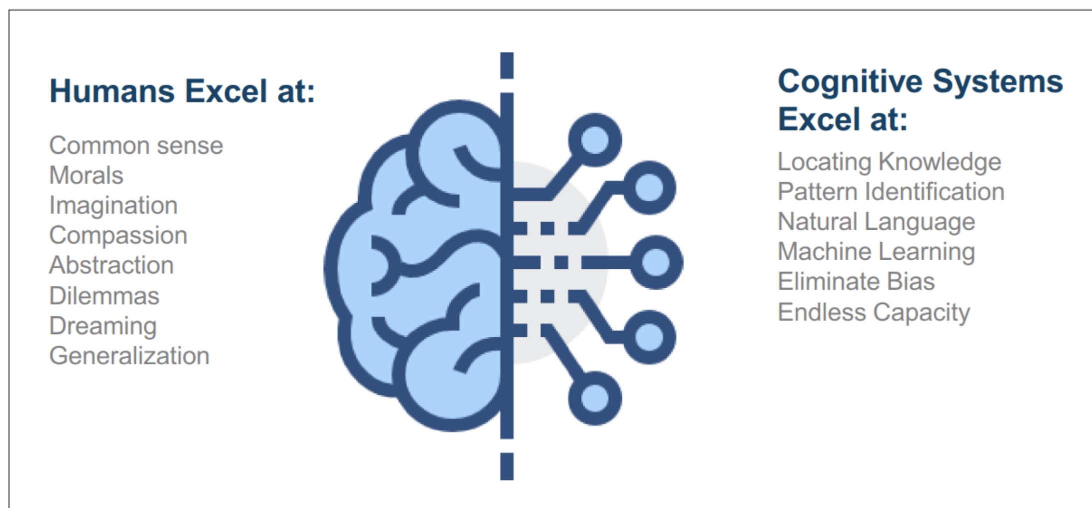

Figura 1.2: Punti di Eccellenza e della macchina

Quindi il computer non deve soltanto prendere ordini, ma deve interagire con l'uomo; durante l'interazione impara immagazzinando dati. La capacità di apprendimento della macchina fa sì che quest'ultima conosca sempre meglio il suo interlocutore.

In particolar modo, il Cognitive Computing, serve per migliorare la comprensione del linguaggio naturale e di generare risposte sempre più vicine ad un contesto di conversazione naturale.

La NLP *("Natural Lenguage Processing")* è il sottocampo dell'informatica, in particolare dell'intelligenza artificiale, che si occupa di comprendere ed elaborare il linguaggio umano.

Le fasi della NLP sono:

- *Elaborazione morfologica*: processo di divisione della frase in token. Quest'ultimi sono simboli che rappresentano parole, numeri, etc.
- *Analisi sintattica*: controllo delle frasi ricevute e successivo controllo per verificare se sono ben formulate.
- *Analisi semantica*: estrapolazione del significato della frase.
- *Analisi pragmatica*: assegnazione del significato della frase in base al contesto.

Tali fasi sono fondamentali per lo sviluppo dei chatbot e degli assistenti virtuali, in quanto consentono di svolgere funzioni sempre più sofisticate, come la prenotazione di un volo o la gestione di un ordine.

In sintesi, l'utilizzo della NLP rappresenta una tappa importante nel progresso dell'IA, in quanto consente ai computer di comprendere e interpretare il linguaggio umano in modo sempre più preciso e naturale. Grazie alla NLP, i chatbot e altri sistemi conversazionali sono in grado di fornire un'esperienza utente sempre più soddisfacente e personalizzata.

1.3 Vantaggi e limiti dei Chatbot

1.3.1 I vantaggi

Il chatbot è una tecnologia diventata sempre più popolare in diversi campi, da quello del customer service fino a quello del marketing e del supporto tecnico. Tale utilizzo però, ci porta a considerare i vantaggi e i limiti che si presentano nel momento in cui si utilizzano nei sistemi. Tra i principali vantaggi, uno dei più importanti è la possibilità di essere disponibile 24 ore su 24, 7 giorni su 7. Questa caratteristica è molto utile quando la sua implementazione è applicata alle aziende che offrono assistenza e supporto ai clienti. Tale caratteristica dà la possibilità ai clienti di ottenere un'assistenza immediata e costante, indipendentemente dai fusi orari e dalle posizioni dei clienti nel mondo.

Un altro grande vantaggio del chatbot è la capacità di gestire un grande volume di richieste simultaneamente. Un persona, di solito, si può occupare solo di una persona alla volta; invece il chatbot rispondendo contemporaneamente a molte richieste, permette di fornire un servizio rapido ed efficiente. Inoltre, potendo memorizzare le informazioni degli utenti, nel tempo è possibile personalizzare le risposte in base a ciò che è stato memorizzato portando ad una migliore esperienza dell'utente. L'aspetto che potrebbe essere più vantaggioso nell'uso del chatbot è quello di poter ridurre i costi aziendali. La sua implementazione nella struttura aziendale è meno costosa rispetto all'assunzione del personale. Tale vantaggio lo si può riscontrare qualora l'azienda che ne fa uso debba compiere operazioni alquanto ripetitive. Un ultimo e non meno importante rispetto ai precedenti vantaggi, è il beneficio dell'aumento della produttività. Potendo memorizzare informazioni, e avendo la posibilità di implementare l'IA per automatizzare diversi compiti ripetitivi e noiosi, il chatobot può liberare la risorsa del personale umano e impegarla per svolgere dei compiti più complessi.

1.3.2 I Limiti

I diversi vantaggi sopraelencati, purtroppo mettono in luce diverse problematache e limiti di cui i chatbot sono affetti.

Il limite più grande, che ad oggi risulta essere ancora una sfida importante per gli studiosi, è la capacità di comprendere il linguaggio naturale e, di conseguenza, capire richeste complesse o ambigue. Tale problematica fa si che i chatbot possano essere facilmente ingannati o manipolati da utenti che cercano di ottenere risposte inesatte o inappropiate.

Questa criticità rischia, anche, di compromettere la sicurezza dei sistemi.

Infatti, chatbot che raccolgono informazioni potrebbero essere vulnerabili agli attacchi informatici e alle perdite di dati.

Un'altra restrizione, seppur meno importante rispetto a quelle precedenti, è l'estrema difficoltà di simulare l'interazione umana. In particolar modo, nel caso in cui un utente voglia cercare un supporto emotivo o voglia risolvere problemi di grande complessità, il chatbot non è in grado di generare risposte adeguate. Ad oggi , infatti, i chatbot risultano impersonali e privi di empatia. Infine, un'altra limitazione dei chatbot è la loro capacità di offrire un esperienza utente personalizzata e altamente interattiva. Questa restrizione è dovuta dal fatto che potrebbero non essere in grado di offrire la stessa gamma di opzioni e di risposte personalizzate che una persona potrebbe fornire in una situazione analoga. Ciò potrebbe portare a non prenderli in considerazione in determinati contesti, come ad esempio, per operazioni in cui si richiede una forte comunicazione con l'interlocutore.

CAPITOLO 2

Strumenti software utilizzati

Nell'ambito dello sviluppo software e dell'intelligenza artificiale l'utilizzo di strumenti software avanzati è fondamentale per creare applicazioni intelligenti e all'avanguardia. Tra questi risaltano, per la loro importanza e versatilità: il framework Rasa, le macchine virtuali e l'ambiente di sviluppo Anaconda. Il framework Rasa rappresenta una soluzione completa per lo sviluppo di chatbot e assistenti virtuali intelligenti; esso è basato sull'intelligenza artificiale e sul machine learning. Inoltre, Rasa offre un set di strumenti e librerie per la creazione di chatbot conversazionali avanzati. Questo framework permette di gestire la comprensione del linguaggio naturale, il dialogo con l'utente: l'integrazione con i sistemi esistenti. Una macchina virtuale è un ambiente software con un alto grado di isolamento e sicurezza che, emulando un computer fisico, consente l'esecuzione di programmi e applicazioni su diverse piattaforme. Questa tecnologia consente agli sviluppatori di creare un software che può essere eseguito su diversi sistemi operativi senza dover riscrivere il codice da zero. Infine, Anaconda rappresenta un ambiente di sviluppo completo per la data science e il machine learning; esso offre un vasto ecosistema di strumenti e librerie preinstallate, semplificando la configurazione di un ambiente di lavoro dedicato all'analisi dati.

2.1 Ambiente di lavoro

L'ambiente di lavoro nello sviluppo software è un elemento cruciale per garantire la produttività, l'efficienza e la qualità del processo di sviluppo. Il computer, utilizzato per la costruzione del chatbot, è un MacBook Pro del 2020 con:

- Processore: Apple Siliccon M1
- Ram: 8 GB
- SSD PCIe 512GB

Il progetto è stato realizzato su una macchina virtuale con il sistema operativo Windows, poiché rappresenta una soluzione versatile e pratica per lo sviluppo software. La scelta di utilizzare questa macchina virtuale è dovuta principalmente al mancato supporto di alcuni componenti software del framework Rasa da parte dell'architettura hardware basata su ARM (Advanced RISC Machines), tecnologia di processori in dotazione ai Apple Mac in commercio dal 2020.

Grazie al suo utilizzo è possibile creare un ambiente isolato all'interno del proprio sistema operativo principale, fornendo un'area dedicata all'esecuzione di applicazioni Windows senza interferire con l'ambiente di lavoro preesistente.

Questa configurazione offre diversi vantaggi, tra cui la possibilità di sperimentare e testare applicazioni su una versione di Windows senza dover installare direttamente il sistema operativo sul proprio computer.

Inoltre, l'uso della macchina virtuale può contribuire alla sicurezza dell'ambiente di lavoro. Essa offre un livello aggiuntivo di protezione che consente di non compromettere la macchina principale grazie all'isolamento dal sistema operativo principale.

Per la progettazione di un software, uno sviluppatore ha bisogno di un IDE (Integrated Development Environment) che fornisca un ambiente integrato per la scrittura del codice. L'IDE offre diverse funzionalità, come l'evidenziazione della sintassi, il completamento automatico del codice, il debugging, etc. Ciò consente agli sviluppatori di lavorare in modo più efficiente ed evita errori di sintassi o logica nel codice.

L'IDE utilizzato per questo progetto è Visual Studio Code (Figura 2.1).

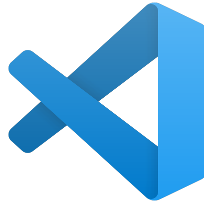

Figura 2.1: Visual Studio Code

Visual Studio Code è un'applicazione multi-piattaforma utilizzabile su diversi sistemi operativi, altamente personalizzabile con una vasta gamma di funzionalità. Essendo accessibile a molti sviluppatori, gli utenti possono configurare l'editor secondo le proprie preferenze; ad esempio, possono impostare scorciatoie da tastiera personalizzate, installare estensioni che aggiungono funzionalità extra, etc. Ciò permette di adattare l'ambiente di lavoro alle specifiche esigenze di sviluppo.

2.1.1 Anaconda

Anaconda (Figura 2.2) è una piattaforma software ampiamente utilizzata nel campo della data science e dell'apprendimento automatico. È stata progettata per semplificare e accelerare il processo di sviluppo e distribuzione di applicazioni basate sull'analisi dei dati.

Figura 2.2: Anaconda

Una delle caratteristiche principali di Anaconda è la sua vasta raccolta di pacchetti open-source, tra cui librerie, framework e strumenti popolari utilizzati per l'analisi dei dati. Esso include librerie come NumPy, Pandas, Matplotlib, SciPy, scikit-learn e molti altri. Anaconda fornisce un sistema di gestione dei pacchetti chiamato Conda, che consente agli utenti di installare, aggiornare e gestire facilmente le dipendenze del software. Essendo un software eseguibile su più piattaforme, esso permette di sfruttare le sue potenzialità su qualsiasi sistema operativo scelto. Il suo punto di forza è la gestione di ambienti virtuali, consentendo agli sviluppatori di creare e gestire facilmente i sandbox di sviluppo. Quest'ultimi permettono di isolare le dipendenze del software, garantendo che un progetto non interferisca con un altro. Ciò è particolarmente utile quando si lavora su progetti con requisiti diversi o quando si vuole mantenere una configurazione stabile per un progetto specifico.

2.2 Rasa

2.2.1 Rasa Framework

Rasa (Rasa [2023]) è un framework open source per la creazione di un chatbot basato sul machine learning supervisionato. Esistono molti altri framework per creare chatbot; i più famosi sono quelli di Google (Dialogflow Google [2023]), Amazon (Amazon Lex Amazon [2023]) e Microsoft (Luis Microsoft [2023]), i quali permettono uno sviluppo rapido e intuitivo del bot attraverso un'interfaccia grafica.

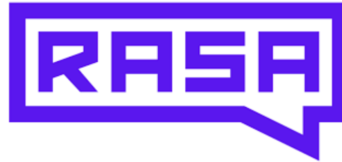

Figura 2.3: Rasa

La scelta di utilizzare Rasa (Figura 2.3) come framework è stata condizionata principalmente dal fatto che esso è open source. Infatti, è possibile intervenire manualmente nel codice in caso di bug e, soprattutto, gode di una community molto ampia, e gli sviluppatori sono sempre pronti a rispondere ad eventuali domande e chiarimenti. Esso si distingue per la sua flessibilità e scalabilità, consentendo agli sviluppatori di personalizzare e adattare facilmente i chatbot alle esigenze specifiche dei loro progetti. Il framework fornisce un'architettura modulare, che permette di integrare diversi componenti, come il riconoscimento del linguaggio naturale (NLU), il sistema di dialogo, l'integrazione di API esterne e molto altro. Una delle sue caratteristiche principali è la capacità di apprendimento continuo. Utilizzando l'apprendimento rinforzato, i chatbot costruiti con Rasa, possono migliorare le loro risposte nel tempo, interagendo con gli utenti e ricevendo feedback. Ciò permette loro di diventare sempre più intelligenti e di offrire esperienze di conversazione sempre migliori.

Con Rasa, è possibile sviluppare chatbot sofisticati e coinvolgenti, fornendo un'esperienza di conversazione di alta qualità agli utenti.

2.2.2 Funzionamento di Rasa

Rasa è composto da due parti: *Rasa Core* e *Rasa NLU* (Figura 2.4).

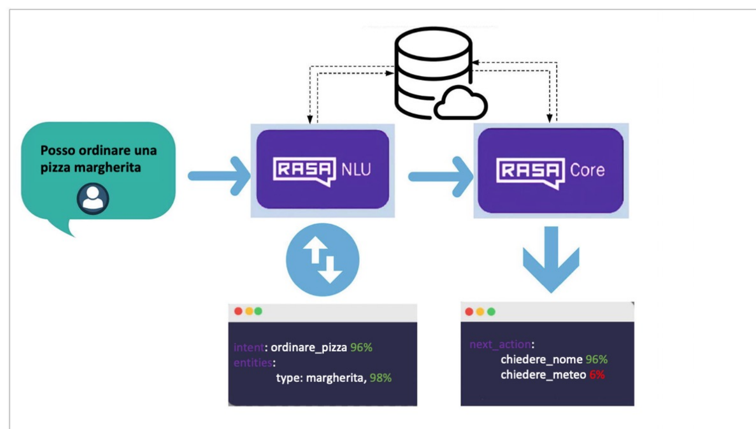

Figura 2.4: Architettura di Rasa

Rasa NLU è la componente che si occupa di estrarre l'intento e le entità dai messaggi forniti dagli utenti. Il processo di tale modulo, può essere suddiviso in due parti principali: l'elaborazione del linguaggio naturale (NLP) e l'estrazione delle informazioni.

1. L'elaborazione del linguaggio naturale si divide in quattro fasi:
   (a) *Tokenizzazione*: il testo ricevuto in input viene suddiviso in unità più piccole chiamate token, come parole o caratteri.
   (b) *Rimozione delle stopwords*: le parole comuni che non contribuiscono significativamente alla comprensione del testo vengono rimosse, ad esempio "il", "e", "di".
   (c) *Lemmatizzazione o stemming*: le parole vengono ridotte alla loro forma base, ad esempio "correndo" diventa "correre".
   (d) *Part-of-Speech (POS) tagging*: viene assegnata una classe grammaticale a ciascuna parola, come sostantivo, verbo, aggettivo, etc.
2. L'estrazione delle informazioni si suddivide in:
   (a) *Intenti*: vengono identificate le intenzioni degli utenti, che rappresentano il loro obiettivo o la loro richiesta. Ad esempio, un intento potrebbe essere "voglio ordinare una pizza" o "trovami delle informazioni sul meteo".
   (b) *Entità*: vengono estratte le informazioni specifiche o le entità rilevanti presenti nel messaggio. Ad esempio, per l'intento "voglio ordinare una pizza", le entità potrebbero includere il tipo di pizza, il nome della pizza, gli ingredienti, etc.

Estratti l'intento e le entità dal messaggio dell'utente, queste informazioni possono essere utilizzate per avviare azioni specifiche all'interno del chatbot.

Utilizzando altri framework, quali **tensorflow**, **spacy** e **sklearn**, Rasa NLU riesce ad analizzare e comprendere la grammatica della frase al fine di classificarla per poter, infine, costruire la risposta opportuna (generata da Rasa Core).

Rasa Core è una componente fondamentale del framework Rasa ed è responsabile della gestione del sistema di dialogo del chatbot. La sua funzione principale è, quella di determinare quale azione eseguire in risposta a un determinato stato di conversazione.

Esso è costituito da uno schema basato su regole e apprendimento rinforzato per gestire il flusso di conversazione. Ciò significa che il sistema di dialogo può essere progettato utilizzando regole predefinite, ma può anche apprendere dalle interazioni con gli utenti per migliorare le risposte future.

Di seguito si riportano i moduli fondamentali delle due componenti precedentemente citate:

- *Rasa Core - Domain:* è un file di configurazione YAML che descrive le caratteristiche del dominio del chatbot. Alcuni elementi chiave che possono essere definiti nel file del dominio includono:
  - intent: intenti di un utente da gestire;
  - entity: informazioni specifiche di un utente;
  - action: azioni che il chatbot può compiere in risposta agli input degli utenti;
  - slot: informazioni di stato che possono essere memorizzate e utilizzate durante la conversazione.

- *Rasa Core - Stories:* utilizzato per definire il flusso delle conversazioni nel chatbot. Le storie rappresentano gli scenari di conversazione che possono verificarsi tra l'utente e il chatbot. Ogni storia è composta da una sequenza di azioni che il chatbot deve eseguire in risposta agli input dell'utente. Le azioni possono essere sia di sistema predefinite (come risposte di saluto o richieste di informazioni) o sia personalizzate dallo sviluppatore nella stesura del codice. Esse consentono di modellare diversi percorsi di conversazione che il chatbot può seguire. La loro definizione consente di addestrare il modello di Rasa sulle diverse possibili interazioni con gli utenti.

- *Rasa Core - Rules:* definisce le regole di dialogo per gestire specifici scenari o comportamenti del chatbot. Le regole vengono formulate per stabilire condizioni e azioni da intraprendere in base agli input dell'utente. Ogni regola è composta da una condizione e da una o più azioni. La condizione può essere un insieme di intenti, entità o altri criteri specifici che devono essere soddisfatti affinché la regola venga attivata. Le regole sono utili per gestire i flussi di conversazione e per fornire un controllo dettagliato e mirato sul comportamento del chatbot. D'altro canto, la definizione di molte regole comprometterebbe la naturale funzione dell'assistente virtuale vincolandone le azioni.

- *Rasa Core - Actions:* sono tutte le operazioni che il chatbot deve intraprendere in risposta agli input dell'utente. Le azioni possono essere di due tipi: azioni di sistema (predefinite) o azioni personalizzate definite dall'utente. Le prime sono già implementate nel framework di Rasa e possono essere utilizzate direttamente senza bisogno di ulteriori personalizzazioni. Per le azioni personalizzate, invece, è possibile eseguire delle specifiche istruzioni con l'ausilio del linguaggio di programmazione Python. Esse vengono implementate come classi Python che estendono la classe *Action* fornita da Rasa. Definire una custom action consente di realizzare chatbot interattivi e dinamici. Per utilizzarle, è necessario configurare un server (*action server*) separato che esegua le azioni custom.

- *Rasa Core - Forms:* è una funzione che consente di gestire le interazioni con l'utente per raccogliere informazioni specifiche in modo strutturato durante una conversazione. Le form in Rasa consentono di creare domande, raccogliere risposte e memorizzare in specifici slot le informazioni fornite dall'utente. Sono spesso utilizzate quando si desidera raccogliere una serie di informazioni da parte dell'utente in modo organizzato e sequenziale, ad esempio per raccogliere dati come il nome, il cognome, la data di nascita, etc. Una volta che tutti gli slot richiesti sono stati riempiti, è possibile eseguire azioni personalizzate.

- *Rasa Core - Slots:* sono delle variabili che rappresentano concetti o informazioni rilevanti per la conversazione. Durante l'interazione con gli utenti, possono essere popolate e aggiornate. Gli slot, dichiarati nel file *Domain*, possono essere definiti come "text" per testo, "categorical" per categorie predefinite o "float" per numeri decimali.

- *Rasa NLU - Intent:* sono le intenzioni che il chatbot può gestire durante una conversazione. Gli Intent vengono utilizzati per identificare e comprendere cosa l'utente sta cercando di comunicare. In particolare, sono etichette che vengono assegnate ai messaggi ricevuti da un utente per classificarli in base alla loro intenzione. Ad esempio, se una persona ha intenzione di ordinare una pizza digitando: "Voglio ordinare una pizza", l'intent associato potrebbe essere "ordinazione_pizza". Nel processo di sviluppo del chatbot, viene creata una lista di intent per il dominio specifico. Questi ultimi vengono definiti con degli *examples* per facilitare la loro comprensione durante la classificazione. Grazie ad essi, il sistema è in grado di comprendere l'obiettivo o il desiderio dell'utente e di reagire di conseguenza.

- *Rasa NLU - Entities:* rappresentano specifiche informazioni estratte dall'input dell'utente durante una conversazione. Le entità sono parti della frase che identificano un valore o un concetto specifico. Ad esempio, se voglio ordinare una pizza e l'utente digita: "Voglio ordinare una pizza margherita e una birra", le entità estratte sono "margherita" come entità "tipologia_pizza" e "birra" come entità "bevanda". Nella fase di progettazione, le entità vengono definite nel dominio specifico e vengono associate a pattern o esempi di frasi che contengono i valori delle entità. Rasa utilizza modelli di apprendimento automatico per identificarle ed estrarle dall'input dell'utente. Il loro uso consente di arricchire la comprensione del contesto estratto dal messaggio dell'utente. Inoltre, le entità possono essere memorizzate negli slots del sistema e utilizzate per personalizzare le risposte o per avviare azioni specifiche basate sui valori acquisiti. Per definire le *Entities*, esistono tecniche per l'estrazione di informazioni dai messaggi degli utenti. Queste tecninche sono:

  - Lookup Table;
  - Regex;

  Le Lookup Table consentono al sistema di "mappare" rapidamente i termini o le frasi comuni ai valori delle entità senza dover utilizzare modelli di machine learning o algoritmi più complessi. Sono particolarmente utili quando si hanno entità con un insieme limitato di valori noti o quando si vogliono estrarre informazioni specifiche in modo preciso. La tecnica Regex è un pattern di ricerca utilizzato per identificare corrispondenze in un testo. Essa viene utilizzata quando si cerca di estrarre informazioni specifiche, come date, numeri di telefono, indirizzi, etc.

2.2.3 Implementazione con altri sistemi

Una delle caratteristiche distintive di Rasa è la sua natura modulare e flessibile, che permette di integrarsi facilmente con altre piattaforme e sistemi. Tra le piattaforme più rilevanti vi sono: Facebook, Telegram e WhatsApp (Figura 2.5).

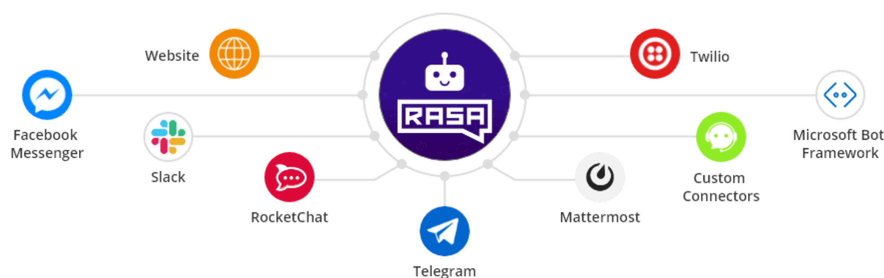

Figura 2.5: Principali Piattaforme con cui si può integrare Rasa

Rasa facilita anche la connessione con le interfacce vocali, consentendo agli utenti di interagire con il chatbot mediante comandi vocali. Ciò rende possibile l'utilizzo di Rasa con assistenti vocali come Amazon Alexa o Google Assistant, ampliandone la portata e l'accessibilità del chatbot. L'ampia gamma di opzioni per la fusione con altre piattaforme, consente di adattarsi con gli ecosistemi esistenti offrendo esperienze utente personalizzate.

2.3 Librerie utilizzate

Nel contesto di un progetto specifico, le librerie software vengono selezionate in base alle esigenze e alle tecnologie utilizzate. Esistono librerie per diversi ambiti, come, ad esempio, l'elaborazione di immagini, l'apprendimento automatico, la gestione dei database, la grafica, la crittografia, e molto altro ancora.

L'uso di librerie software consente di risparmiare tempo nello sviluppo di un progetto, in quanto molte funzionalità complesse sono già implementate e testate. Inoltre, le librerie offrono una comunità di supporto attiva, con documentazione, esempi e forum di discussione dove gli sviluppatori possono condividere esperienze e risolvere eventuali problemi. Per la realizzazione del chatbot le librerie utilizzate sono:

- Rasa_Sdk;
- Pandas;
- Tensorflow;
- Datasketch;
- Difflib.

Rasa SDK (Software Development Kit) è una libreria Python fornita da Rasa che consente di creare e personalizzare le azioni del chatbot (Figura 2.6).

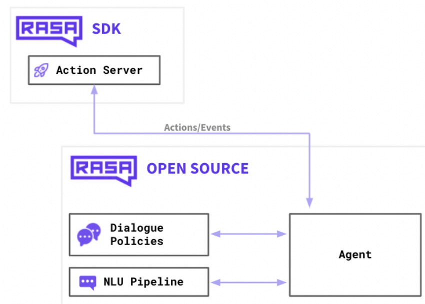

Figura 2.6: Rasa Sdk

Esso offre una serie di funzionalità per definire il comportamento del chatbot e per gestire l'interazione con gli utenti. Oltre a ciò, tale libreria fornisce un ambiente di sviluppo e un server per le azioni, che semplificano il processo di sviluppo, la fase di test e la distribuzione delle azioni personalizzate. Il server permette di eseguire le azioni in un ambiente separato e di comunicare con il core di Rasa attraverso un'API (Application Programming Interface). Quest'ultima rappresenta uno strumento essenziale per lo sviluppo di chatbot customizzati con Rasa.

Pandas è una potente libreria open-source per la manipolazione e l'analisi dei dati in Python (Figura 2.7).

Figura 2.7: Pandas

È ampiamente utilizzata nel campo della data science grazie alla sua efficace gestione di strutture dati tabulari, come i dataframe. La sua particolarità è quella di fornire strutture dati flessibili e adatte alla gestione di elementi eterogenei, come, ad esempio, file CSV, fogli di calcolo Excel, database SQL, e altro ancora. Il principale oggetto offerto da Pandas è il dataframe, che rappresenta una tabella con righe e colonne, simile a un foglio di calcolo. I dataframe di Pandas consentono di manipolare, filtrare, aggregare e analizzare i dati in modo efficiente.

Pandas mette a disposizione un'ampia gamma di funzionalità per lavorare con i dati, inclusi metodi per la pulizia, la trasformazione, il calcolo delle statistiche, la gestione dei valori mancanti, il merging, la concatenazione, l'indicizzazione e la selezione dei dati, e molto altro ancora. Il suo punto di forza è la sua capacità di gestire grandi quantità di informazioni in modo efficiente, grazie alla sua implementazione ottimizzata dei dati in memoria. Ciò consente di elaborare e analizzare dataset di grandi dimensioni senza problemi di prestazioni.

TensorFlow (Figura 2.8) è una libreria open-source per il machine learning e il calcolo numerico sviluppata da Google. È una delle librerie più popolari utilizzate per creare e addestrare modelli di intelligenza artificiale. Le reti neurali sono un modello computazionale rappresentano uno dei modelli computazionali più rilevanti in cui viene applicato TensorFlow.

Figura 2.8: TensorFlow

TensorFlow offre una vasta gamma di funzionalità per la creazione e l'addestramento di modelli per il machine learning. Il concetto di base è il "grafo computazionale", dove le operazioni matematiche vengono organizzate in un grafo orientato, che rappresenta il flusso di calcolo. Ciò consente di eseguire operazioni di calcolo in parallelo e sfruttare sistemi per l'accelerazione dell' hardware, come le GPU, per ottenere prestazioni ottimizzate.

Una delle caratteristiche distintive di questa libreria è la sua flessibilità e scalabilità. Essa rappresenta la creazione di modelli di machine learning sia per task di apprendimento supervisionato che non supervisionato.

Diventato uno standard nell'industria del machine learning, è utilizzato per sviluppare applicazioni di intelligenza artificiale, come il riconoscimento di immagini, il trattamento del linguaggio naturale, la traduzione automatica, la robotica, e molto altro ancora.

Datasketch (Figura 2.9) è una libreria Python open-source che fornisce strumenti per la creazione di strutture dati probabilistiche e per l'analisi approssimata di grandi insiemi di dati. È particolarmente utile quando si lavora con dati di grandi dimensioni e si desidera ottenere una stima approssimata di diverse metriche, come la similarità, la cardinalità, la presenza di elementi comuni, etc.

Figura 2.9: Datasketch

Esso è ampiamente utilizzato per il data mining, l'analisi dei dati, la ricerca di testo, l'elaborazione del linguaggio naturale, e altro ancora. La sua flessibilità lo rende una scelta popolare per la gestione approssimata dei dati in situazioni dove la precisione esatta non è fondamentale, ma è richiesta una buona scalabilità ed efficienza.

Difflib è una libreria Python che fornisce strumenti per confrontare sequenze di dati. È particolarmente utile per calcolare le differenze tra due stringhe o sequenze e, per generare output come patch o liste di operazioni di modifica.

CAPITOLO 3

Specifica e analisi dei requisiti

Nel capitolo in esame, si affronta un progetto sviluppato per il Sindacato Italiano Militari Carabinieri. Verrà affrontata l'implementazione di un chatbot destinato al sito web dell'organizzazione. L'obiettivo principale del chatbot è quello di fornire assistenza e semplificare la ricerca dei documenti. A questo scopo, è stata avviata una fase di studio approfondito per comprendere i requisiti necessari a garantire un prototipo funzionante e operativo del chatbot.

3.1 Contesto di utilizzo

Il Sindacato Italiano Militari Carabinieri (SIM) è un'organizzazione sindacale italiana che rappresenta gli interessi dei militari dell'Arma dei Carabinieri: una delle forze armate italiane (Figura 3.1).

Figura 3.1: Sindacato Italiano Militari Carabinieri

Tra i ruoli principali che il SIM ricopre troviamo:

- attività di negoziazione, rappresentanza e assistenza legale con lo scopo di tutelare i diritti e le condizioni di lavoro dei militari dell'Arma dei Carabinieri;
- attività di promozione di dialogo tra i militari dell'Arma dei Carabinieri e le istituzioni competenti, come ad esempio, il Ministero della Difesa, il Comando Generale dei Carabinieri, etc.;
- perfezionamenti riguardanti orari di lavoro, remunerazione, promozioni, regimi pensionistici, norme di sicurezza e altri elementi connessi riguardanti la professione militare.

Il SIM offre diverse modalità di supporto ai membri dell'Arma dei Carabinieri, attraverso l'utilizzo di un portale internet dedicato. Questa piattaforma web si compone di una sezione pubblica, aperta a tutti gli utenti, e di una sezione riservata ai soli membri dell'Arma dei Carabinieri. Quest'ultima sezione fornisce, agli agenti dell'arma, una vasta gamma di informazioni personalizzate per soddisfare le loro specifiche richieste.

Al fine di ottimizzare l'esperienza dell'utente sul portale e garantire un supporto rapido e affidabile, il sindacato ha incaricato Fermo Tech della creazione di un chatbot personalizzato.

FermoTech (Figura 3.2) è la piattaforma collaborativa Fermana della Regione Marche, centro tecnologico di eccellenza, che mira a una collaborazione sinergica ed efficace tra mondo accademico e industria.

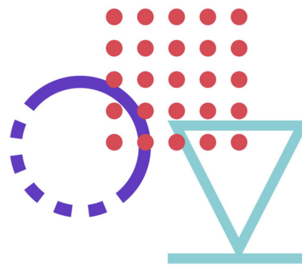

Figura 3.2: Fermo Tech

L'obiettivo del chatbot da progettare è quello di migliorare l'assistenza fornita dal sito internet, senza la necessità di coinvolgere un operatore umano. In questo modo si riuscirebbe ad offrire una soluzione più efficiente e pratica.

Analogamente alla piattaforma web, il SIM ha definito che anche il chatbot dovrà avere una sezione accessibile a tutti gli utenti e una sezione con funzionalità specifiche riservata agli utenti che si sono autenticati nell'area riservata.

3.1.1 Sezione pubblica

In questa sezione verranno presentate le funzioni del chatbot accessibili agli utenti che si collegano al sito web. Nella fase iniziale dello sviluppo del chatbot, l'obiettivo principale è quello di fornire supporto per operazioni di base. Le attività che il chatbot deve gestire, indipendentemente dal tipo di utente che accede al sito, comprendono:

- l' iscrizione al sito
- il recupero della Password
- la ricerca del Segretario di Regione

Sebbene tutti gli utenti possano usufruire di queste attività, le prime due possono essere completate solo dai carabinieri. Infatti, l'iscrizione al portale può essere completata solo dagli agenti dell'Arma dei Carabinieri, poiché, nella relativa procedura, vengono richieste informazioni che possiede soltanto chi è un membro dell'arma. Analogo discorso vale anche per il recupero della password, dal momento che questo tipo di funzione è conseguente alla prima.

La funzione di iscrizione al sito consiste in una procedura che fornisce le istruzioni necessarie su come effettuare la registrazione al portale da parte dell'utente. In risposta alla richiesta di iscrizione, il chatbot fornisce una guida esaustiva sui passaggi che il militare dovrà seguire per registrarsi.

La procedura di iscrizione è definita da una risposta composta dai seguenti tre messaggi:

- *Gentile collega, in questa pagina puoi effettuare la procedura d'iscrizione online e i necessari documenti verranno automaticamente generati dal nostro sito web ed inviati alla tua mail.*
- *Per completare l'iscrizione è poi necessario, una volta che avrai firmato la delega, che tu faccia l'Upload della stessa, della copia fronte/retro del tuo documento di identità in corso di validità e della tessera sanitaria in caso non sia in possesso della nuova Carta di Identità elettronica. Scarica delega[link_al_sito]*
- *L'iscrizione ha validità dal 1° giorno del mese successivo all'iscrizione fino al 31 dicembre di ogni anno e si intende tacitamente rinnovata ove non venga revocata dall'interessato entro il 31 ottobre. Grazie per la fiducia accordataci.*

Un ragionamento simile è stato svolto per la funzione di recupero password. Anche in questa situazione, vengono forniti sei messaggi di risposta, i quali contengono le istruzioni per la procedura di reset della password, così delineati:

- *Per ripristinare la password andare sulla propria area riservata. I passi da eseguire sono:*
- *1) Andare nell'Area utenti [link_al_sito]*
- *2) Cliccare sull link password dimenticata*
- *3) Successivamente dovrai inserire la tua mail e cliccare il tasto Invia Richiesta*
- *4) Controllare sulla casella di posta [inserita nel punto 3] l'arrivo di una nuova comunicazione. Potrebbe essere necessario controllare anche nell'area SPAM della propria casella di posta elettronica*
- *5) Aprire la mail e cliccare sul link per il recupero della password*
- *6) Inserire la nuova password*

Per quanto riguarda la funzione di ricerca del segretario di regione, il compito del chatbot è quello di fornire le informazioni necessarie riguardanti il segretario di regione che l'utente desidera cercare.

In dettaglio verranno comunicati i seguenti dati:

- Nome;
- Cognome;
- Indirizzo;
- Numero di Cellulare;
- E-mail.

3.1.2 Sezione privata

A seguito dell'autenticazione di un utente, si accede dalla sezione pubblica a quella privata. La sezione privata del sito, oltre alle funzioni della sezione pubblica, offre funzionalità specifiche per gli utenti autenticati. Queste funzioni sono riservate esclusivamente agli utenti registrati al sito. Il chatbot in esame, così come per la sezione publica, dovrà fornire supporto anche alla navigazione della sezione privata.

Le funzioni specifiche della sezione privata sono:

- procedura per il trattamento Pensionistico e Stipendiale;
- ricerca di un allegato;
- gestione della polizza e dei sinistri.

La prima funzione consiste nel fornire all'utente le indicazioni necessarie per accedere alle informazioni relative al trattamento pensionistico e stipendiale. In questa modalità, il chatbot risponde alla richiesta dell'utente fornendo le informazioni desiderate, sia per il trattamento pensionistico che per quello stipendiale.

Alla richiesta delle informazioni inerenti il primo trattamento, l'assistente risponderà con sette messaggi:

1. *Ecco il modulo da scaricare [link_al_modulo]*
2. *Successivamente nella pagina [link_al_sito]*
3. *Allegare la documentazione:*
4. *a) Modulo richiesta consulenza previdenziale*
5. *b) Estratto conto contributivo INPS (esclusivamente in formato .xml)*
6. *c) Ultimo statino stipendiale*
7. *E premere il tasto Invia*

Per quanto riguarda il secondo trattamento, il chatbot fornirà una risposta composta da sei messaggi:

1. *Ecco il modulo da scaricare[link_al_modulo]*
2. *Successivamente nella pagina[link_al_sito]*
3. *Allegare la documentazione:*
4. *a) Modulo richiesta consulenza stipendiale*
5. *b) Ultimo statino stipendiale*
6. *E premere il tasto Invia*

Per quanto concerne la ricerca di un allegato, il chatbot deve essere altamente efficace nel supportare l'utente nella ricerca di documenti all'interno del sito, quando necessario. Per questa funzione, il SIM ha dato totale libertà creativa allo sviluppatore.

In ultimo, per quanto riguarda la gestione della polizza e dei sinistri, il chatbot fornirà assistenza tramite messaggi e immagini, per guidare correttamente l'utente autenticato lungo il processo di compilazione della procedura.

I passaggi che il chatbot descriverà sono:

- *Step 1: Accedi al sito.*
- *Step 2: Entra nella tua area personale e clicca su tutela legale situato sotto all'immagine della tessera*
- *Step 3: A questo punto si aprirà un sito. In alto clicca sul pulsante denuncia un sinistro*
- *Step 4: Scarica il modulo cliccando sul pulsante Download oppure cliccando qui sotto.*
- *[link_al_sito]*
- *Step 5: Successivamente compila il modulo e poi invialo alle email:*
- *[mail]*
- *[mail]*

3.2 Analisi dei requisiti

Dopo aver stabilito le direttive relative ai compiti del chatbot, il primo passo è stato quello della raccolta dei requisiti, per valutarne la fattibilità. Dopo un'attenta analisi delle attività, si è constatato che gran parte delle richieste poteva essere soddisfatta mediante adeguata istruzione sugli intenti e sulle azioni da voler eseguire.

Invece, per quanto concerne due delle funzioni descritte, ovvero la ricerca del segretario di regione e la ricerca di un allegato, si sono riscontrati due ostacoli principali.

Il primo ostacolo è la mancanza di disponibilità delle informazioni necessarie alla creazione delle due funzioni; per tale ragione, è stato richiesto al SIM un metodo per la raccolta di tali informazioni. Nella ricerca di una metodologia per l'acquisizione dei dati, il SIM ha reso noto che le informazioni di cui si necessitava, sono ospitate su un server gestito da un'azienda esterna. Considerando che questo progetto mira a valutare la fattibilità e l'utilità del chatbot, si è deciso di non effettuare una connessione diretta a questo database, non gestito direttamente dal SIM; invece, è stato creato un file esterno che fungesse da rappresentazione del database stesso.

Il secondo ostacolo riguarda il fatto che le funzioni del chatbot si suddividono tra quelle utilizzabili nella sezione pubblica e quelle utilizzabili nella sezione privata. Si è affrontato, perciò, il problema relativo alla gestione di questa distinzione. È evidente che solo gli utenti che si sono autenticati nel sito possono accedere alla sezione privata. Poiché non si ha accesso diretto al server, è stato adottato un diverso approccio, ovvero quello di sviluppare due chatbot separati, uno per la sezione pubblica e uno per la sezione privata. In questo modo, il chatbot relativo alla sezione privata può essere interrogato soltanto se un utente si è autenticato. Questa scelta è stata fatta per garantire la privacy delle funzioni.

CAPITOLO 4

Progettazione

Questo capitolo si focalizza sulle dinamiche e sui processi decisionali che hanno guidato alla progettazione ottimale del chatbot, ponendo particolare attenzione sulla sua velocità ed efficienza. Si inizierà descrivendo l'installazione della macchina virtuale e del chatbot e l'ambiente di lavoro, passaggio fondamentale per consentire lo sviluppo del chatbot. Nei paragrafi successivi, si approfondiranno le idee e gli studi effettuati, offrendo una panoramica completa del processo di costruzione del chatbot. Saranno esposti dettagliatamente gli intenti, la pipeline utilizzata, le regole e le azioni che il chatbot esegue durante il suo utilizzo.

4.1 Installazione e Configurazione

Il computer utilizzato per lo sviluppo del chatbot è un Macbook Pro del 2020, che presenta un'architettura innovativa e completamente rinnovata rispetto alle tradizionali soluzioni di elaborazione dati offerte dalle architetture Intel e AMD. Apple Silicon, infatti, rappresenta una serie di system-on-a-chip (SoC) [Wikipedia [2023]] sviluppati da Apple specificamente per i propri dispositivi (Figura 4.1).

Figura 4.1: Architettura M1 (ARM)

I SoC Apple contengono al loro interno una serie di componenti a cui viene attribuito uno scopo specifico:

- CPU: coprocessore responsabile dell'elaborazione sequenziale, delle istruzioni aritmetiche e di calcolo in modo seriale;
- GPU: coprocessore dedicato all'elaborazione parallela delle istruzioni grafiche tridimensionali;

- NPU: coprocessore appositamente progettato per il Neural Engine, consentendo l'apprendimento integrato (machine learning) e l'uso della realtà aumentata;
- ISP: coprocessore responsabile delle elaborazioni delle immagini catturate dal sensore fotografico;
- Mx: coprocessore che gestisce la raccolta dei dati provenienti dai sensori integrati, come l'accelerometro, il giroscopio, la bussola e il barometro;
- SEP: coprocessore incaricato della protezione dei dati personali attraverso l'utilizzo di chiavi crittografate.

A causa delle sue particolari caratteristiche, diversi software hanno incontrato notevoli difficoltà di esecuzione su questa nuova architettura sviluppata da Apple. Di conseguenza, l'azienda, ha provveduto a fornire un emulatore nativo che permettesse l'esecuzione dei software sviluppati per le architetture esistenti anche sulla nuova macchina appena introdotta. Il software di emulazione è Rosetta 2. Esso è un software di traduzione binaria sviluppato da Apple e consente di tradurre dinamicamente le istruzioni del codice dell'applicazione Intel in istruzioni compatibili con l'architettura Apple Silicon, consentendo, così, alle app Intel-based di funzionare in modo trasparente sui nuovi dispositivi Apple. Questo aiuta a garantire la compatibilità e la continuità delle applicazioni durante la transizione verso l'architettura Apple Silicon.

Nonostante l'utilizzo di questo traduttore, in alcuni casi non era garantita la piena compatibilità all'architettura ARM per alcuni software, portando all'eventuale compromissione dell'esecuzione di determinati servizi o componenti e causando potenziali errori durante l'utilizzo.

Purtroppo, durante l'installazione di Rasa sul computer con questa specifica architettura, si è riscontrato un problema. Nonostante lo sviluppo del software Rasa sia esteso ai tre sistemi operativi più diffusi (Windows, Linux, Apple), l'architettura ARM e Rosetta 2 hanno presentato limitazioni nell'esecuzione di alcuni componenti cruciali per il suo funzionamento, in particolare per la libreria TensorFlow.

Per superare questa problematica e sfruttare appieno Rasa, si è optato per l'installazione di una macchina virtuale. La virtualizzazione del sistema operativo, in questo caso Windows, e l'emulazione di una scheda video VGA (Video Graphics Array) hanno consentito di utilizzare tutte le funzionalità di Rasa in modo completo.

4.1.1 Installazione della macchina virtuale

In questa sezione, verranno illustrati i passaggi effettuati e le configurazioni apportate per garantire un'installazione corretta dell'ambiente virtuale.

Il primo passo consiste nella selezione della macchina virtuale. Durante la fase di progettazione sono state prese in considerazione diverse opzioni, tra cui Parallels e UTM, in quanto capaci di emulare un sistema operativo in modo adeguato e di funzionare correttamente sull'architettura ARM. È importante sottolineare che, nonostante esistano diverse soluzioni commerciali per l'emulazione di macchine virtuali su macOS, durante la fase di progettazione non si è avuta la piena compatibilità con l'hardware specifico utilizzato nel progetto.

Tra Parallels e UTM, la scelta è ricaduta su UTM, un'opzione open source e gratuita (Figura 4.2).

Figura 4.2: Il logo di UTM

UTM impiega il framework di virtualizzazione Hypervisor di Apple per eseguire sistemi operativi ARM su Apple Silicon a velocità quasi native. Inoltre, è disponibile un'emulazione a prestazioni inferiori per eseguire x86/x64 (Processori a 32 bit/64 bit) su Apple Silicon.

Una volta seleziona il software da utilizzare per l'emulazione della macchina virtuale, per poterlo utilizzare sono stati necessari i seguenti passaggi e le seguenti configurazioni.

1. Scaricare il file dal sito web del proprietario del software.

2. Avere a disposizione un sistema operativo in formato ISO (Estensione utilizzata per i file di immagine, cioè un file che contiente tutti i metadata di un archivio). In questo caso si avrà a disposizione il sistema operativo Windows 10 nel suddetto formato.

3. Avviare la procedura di installazione cliccando due volte sul file appena scaricato. All'apertura si avvierà una schermata dove bisognerà trascinare l'icona nella cartella delle applicazioni. Con quest'ultima procedura UTM sarà installato con successo.

4. Aprire il software UTM. Al primo avvio comparirà una schermata (Figura 4.3). Successivamente premere sul tasto *Create a New Virtual Machine* evidenziato in rosso nella figura sottostante.

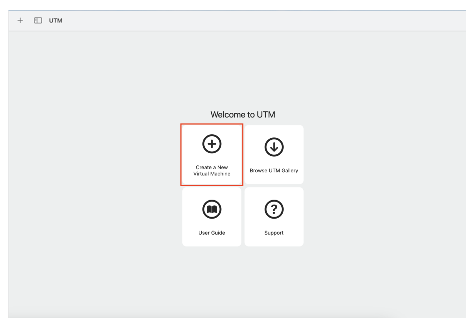

Figura 4.3: UTM: Crezione macchina virtuale

5. Selezionare l'opzione *Virtualize* (Figura 4.4).

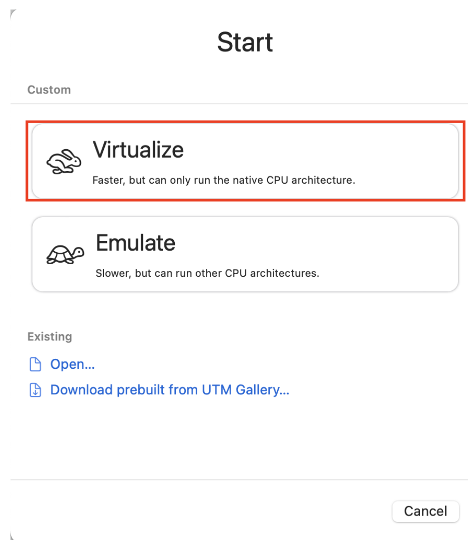

Figura 4.4: UTM: Scelta della tipologia di virtualizzazione

6. Selezionare il sistema operativo da virtualizzare (nel nostro caso sceglieremo Windows) (Figura 4.5).

Figura 4.5: UTM: Scelta del sistema operativo da virtualizzare

7. Spuntare la voce *Install Windows 10 or higher* e la voce *Install drivers and SPICE tools* (Punto 1 e 2 della Figura 4.6). Successivamente, premendo il tasto *Browse* (Punto 3 della Figura 4.6), si aprirà una finestra dove sarà possibile selezionare il file ISO del sistema operativo. Selezionato il file d'immagine, bisognerà cliccare sul pulsante *Continue*.

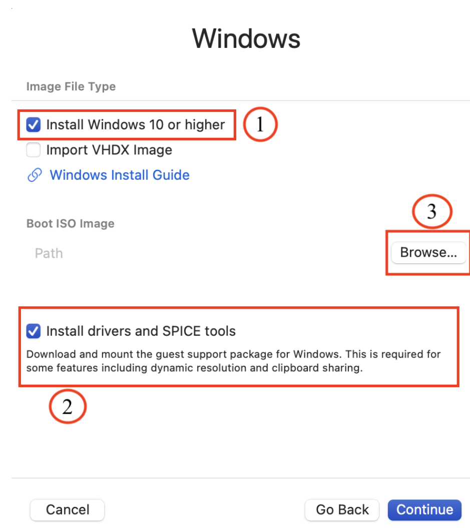

Figura 4.6: UTM: Configurazione dell'avvio dell'installazione di Windows 10

8. In questa schermata (Figura 4.7) si potrà selezionare l'hardware (virtuale) da allocare per la macchina virtuale. Come da Figura 4.7, è stato deciso di allocare 4096MB (4GB) di RAM e 4 processori.

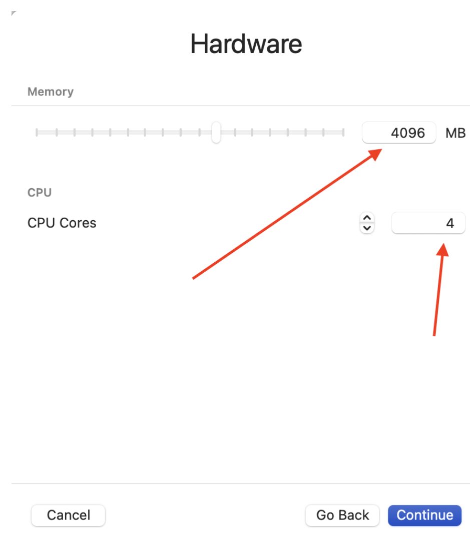

Figura 4.7: UTM: Configurazione della quantità di RAM e dei processori per Windows 10

9. Scegliere la quantità di hard disk da allocare alla macchina e cliccare il tasto *Continue* (Figura 4.8). Per la scelta dei GB da allocare alla macchina virtuale, qualsiasi utente può scegliere autonomamente la quantità da assegnare rispettando sempre i requisiti minimi dichiarati da Windows per una corretta installazione.

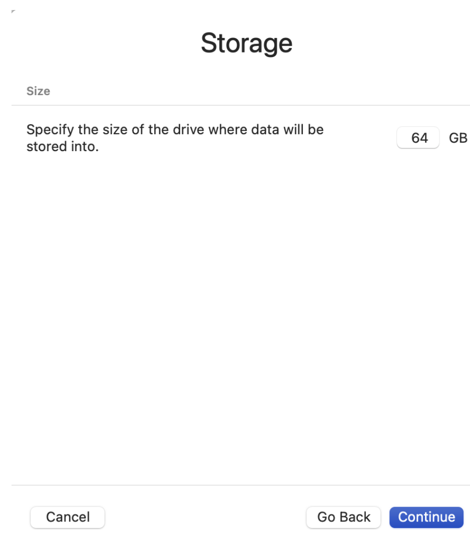

Figura 4.8: UTM: Configurazione della quantità di Hard Disk da allocare a Windows 10

10. Nell'immagine sottostante (Figura 4.9) sono riportati gli ultimi due passaggi per la creazione della macchina. Nella parte a sinistra dell'immagine (Punto 1), è possibile selezionare una cartella da condividere con il proprio computer. La condivisione di una cartella permette di avere un "canale" di comunicazione tra il proprio personal computer e la macchina virtuale, per lo scambio di file. Nella parte di destra (Punto 2) il software riepiloga tutti gli step effettuati fino ad adesso. Effettuato un ultimo checkup, cliccare sul tasto *Save*.

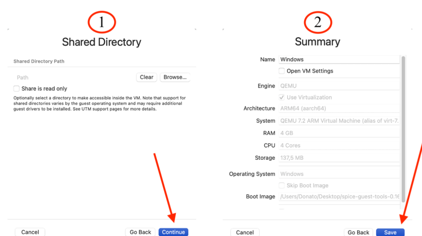

Figura 4.9: UTM: Scelte opzionali e conclusione della configurazione

11. La macchina è stata creata con successo e si troverà nella parte sinistra della schermata iniziale del software UTM (Figura 4.10). Successivamente, prima di avviare la procedura di installazione del sistema operativo, sono state apportate delle ulteriori configurazioni. Per poterle effettuare, bisogna premere con il tasto destro del mouse sulla macchina virtuale appena creata e, successivamente, sul tasto *Edit* (Figura 4.10).

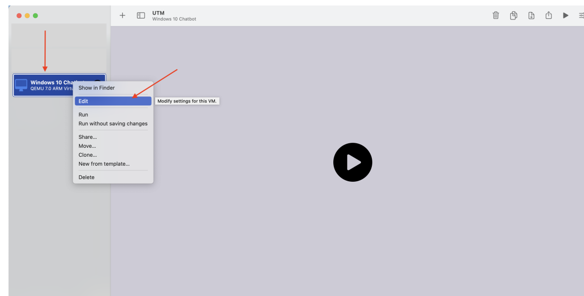

Figura 4.10: UTM: Macchina Virtuale

12. Verrà aperta una finestra dove sarà possibile effettuare diversi setting sulla propria macchina virtuale. Per il progetto in questione, sono state apportate due ulteriori configurazioni (Punto 1 e Punto 2 della Figura 4.11).

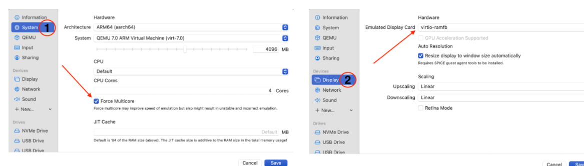

Figura 4.11: UTM: Configurazioni personalizzate

13. Infine, avviare la macchina virtuale premendo sul tasto *play* collocato alla destra del nome della macchina (Figura 4.10). Successivamente, si aprirà una finestra che darà inizio all'installazione del sistema operativo desiderato (in questo caso, Windows 10). Si sorvoleranno i dettagli dei passaggi per l'installazione di Windows 10.

4.1.2 Installazione di Rasa

In questa sezione verranno illustrati i passaggi effettuati e le configurazioni apportate per garantire un'installazione corretta del software Rasa.

Prima di procedere con l'installazione di Rasa, è necessario installare il software Anaconda. La procedura di installazione di Anaconda è molto semplice e consiste nello scaricare il file di installazione dal sito web ufficiale di Anaconda, avviare il programma di installazione, facendo doppio clic sul pacchetto appena scaricato e seguire le istruzioni fornite durante l'installazione. Non verranno riportati qui i dettagli dei passaggi in quanto la procedura è intuitiva e guidata.

Dopo aver completato l'installazione di Anaconda, il primo passo per installare Rasa è aprire il prompt dei comandi di Anaconda. Da questo punto in avanti, la procedura di installazione richiederà l'utilizzo di alcuni comandi da eseguire tramite il terminale (Figura 4.12). Durante l'esecuzione di questi comandi, è importante evidenziare che sarà richiesta una conferma per procedere, bisognerà rispondere sempre con la lettera *"y"* e premere invio per confermare l'azione.

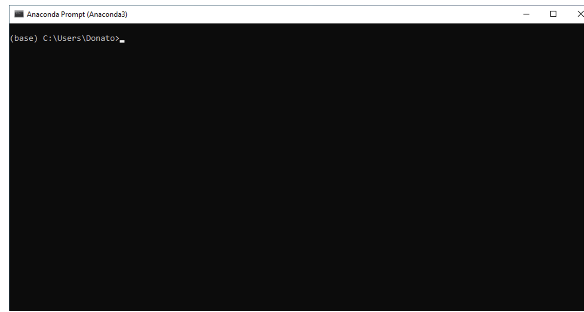

Figura 4.12: Prompt dei comandi di Anaconda

Di seguito sono elencati i comandi fondamentali per un'installazione corretta:

1. Per la creazione di una variabile *env* digitare il seguente comando (Figura 4.13 nell'immagine la variabile è *pippo*):

   conda create -n [nome variabile] python=3.8

   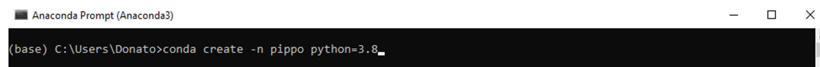

   Figura 4.13: Comando per la creazione della variabile *env*

   La variabile *env* in Anaconda viene utilizzata per specificare l'ambiente virtuale su cui eseguire determinate operazioni, come l'installazione o l'esecuzione di pacchetti o script Python. L'utilizzo degli ambienti virtuali è utile per gestire in modo efficiente le dipendenze dei pacchetti e mantenere un ambiente di sviluppo pulito e isolato per progetti diversi. Ogni ambiente virtuale ha la propria versione di Python (nel caso di questo progetto si utilizza la Versione 3.8) e i pacchetti preinstallati; questo consente la coesistenza di diverse configurazioni, senza interferenze tra loro.

2. Attivare la variabile *env* appena creata, digitando il seguente comando (Figura 4.14 nell'immagine la variabile è *pippo*):

   conda activate [nome variabile]

   Figura 4.14: Comando per l'attivazione della variabile *env*

   Per verificare che la variabile da noi creata sia correttamente attivata, bisogna guardare alla sinistra del percorso dove il prompt sta lavorando. Infatti, come si evince dalla Figura 4.15, non verrà più visualizzata la parola *(base)* accanto al percorso ma sarà cambiata con la variabile *env* creata per il progetto.

Figura 4.15: Verifica dell'attivazione della variabile *env*

3. Riconfigurare il gestore dei pacchetti *pip* digitando:

   python -m uninstall pip

   Il comando *pip* è uno strumento di linea di comando utilizzato per la gestione dei pacchetti Python. È un acronimo che sta per "Pip Installs Packages". Il comando *pip* è ampiamente utilizzato nella community Python per gestire le dipendenze tra i pacchetti; esso permette agli sviluppatori di installare e utilizzare librerie di terze parti nel proprio ambiente di sviluppo.

4. Installare la versione di *pip* in funzione della versione di Python installata digitando:

   python -m pip install -U pip

5. Avviare l'installazione di Rasa, scrivendo:

   pip install rasa

6. Al termine dell'installazione di Rasa, è necessario installare le cartelle principali, necessarie al primo avvio del progetto. Per creare tali cartelle, è sufficiente eseguire il comando seguente:

   rasa init

7. Dopo aver inviato il comando precedente, verrà richiesto di selezionare il percorso in cui creare i file necessari per il chatbot. Si può scegliere di mantenere il percorso corrente o selezionarne uno alternativo, inserendo il percorso desiderato direttamente nel terminale. Nel caso di questo progetto, come percorso di destinazione si è scelto di utilizzare una cartella situata sul desktop.

8. Al termine della creazione dei file, l'utente verrà invitato ad eseguire il primo addestramento del modello del chatbot e, successivamente, ad avviarlo. Tuttavia, poiché Rasa crea un semplice chatbot di esempio, durante la creazione dei file per il progetto è possibile eseguire entrambe le operazioni fin dal primo avvio. Si può scegliere anche di rispondere *no* a entrambe le richieste.

4.2 Scelta del modello NLP

In questo paragrafo si intende fornire una chiara esposizione delle ragioni che hanno portato all'adozione del DIETClassifier e delle sue molteplici potenzialità, che lo hanno reso una scelta rilevante e fondamentale per il progetto in questione. DIETClassifier è uno dei componenti chiave del framework di Rasa per l'elaborazione del linguaggio naturale. Questo classificatore avanzato utilizza una combinazione di tecniche di deep learning per l'analisi del testo e per l'etichettatura degli intenti e delle entità. Sfruttando una rete neurale a più livelli, DIETClassifier è in grado di apprendere rappresentazioni linguistiche complesse, consentendo una classificazione precisa e affidabile dei messaggi.

Attraverso l'addestramento su un ampio corpus di dati annotati, il modello acquisisce la capacità di comprendere il contesto, identificare gli intenti degli utenti e riconoscere le entità rilevanti all'interno di una conversazione. Grazie alla sua flessibilità e alle prestazioni elevate, DIETClassifier rappresenta uno strumento fondamentale per la creazione di chatbot

intelligenti e interattivi, in grado di comprendere e rispondere in modo accurato alle richieste degli utenti.

Durante l'addestramento del modello, o durante la sua fase di produzione, quando si tenta di identificare l'*intento* e le *entità* associati a un'interazione utente, i dati attraversano una pipeline che esegue una sequenza di operazioni. Sebbene non esista un limite tecnico ai tipi di operazioni possibili, tre sono i passaggi comuni alla maggior parte delle pipeline: tokenizzazione, featurizzazione e addestramento o inferenza.

Di seguito verrà fornita una breve spiegazione di tali passaggi, approfondendo ulteriormente:

- *Tokenizzazione*

  La tokenizzazione consiste nell'ottenere l'elenco di parole (token) che compongono una frase. Ad esempio, la frase "Buongiorno sono il tuo primo chatbot" verrebbe tokenizzata come:

  ```
  "Buongiorno", "sono", "il", "tuo", "primo", "chatbot"
  ```

  Le caratteristiche di molti modelli derivano da singole parole, piuttosto che da intere frasi. I token ottenuti da questo processo saranno successivamente utilizzati nella pipeline per l'estrazione delle feature.

- *Caratterizzazione*

  La caratterizzazione è il processo di trasformazione delle parole in numeri significativi (o vettori) che possono essere inseriti nell'algoritmo di addestramento.

- *Addestramento / Inferenza*

  Durante la fase di addestramento, l'algoritmo impara dalle feature derivanti dai dati di testo non elaborati. Durante l'inferenza (quando il modello addestrato viene utilizzato per fare previsioni), i dati grezzi seguono lo stesso percorso. Le espressioni vengono tokenizzate, le feature vengono estratte e utilizzate per prevedere l'intento e le entità.

4.3 Progettazione chatbot

Dopo aver presentato le fasi di installazione, configurazione e selezione del modello di elaborazione del linguaggio naturale (NLP) utilizzato nella progettazione del chatbot, questa sezione si focalizzerà sulla descrizione dettagliata dei passaggi e delle analisi effettuate durante la progettazione del chatbot per il Sindacato Italiano Militari dei Carabinieri.

Prima di iniziare qualsiasi analisi, è essenziale configurare attentamente il file di configurazione di Rasa, noto come *config.yml*. Questo file gioca un ruolo fondamentale nella definizione della logica di comprensione del chatbot e fornisce una guida concettuale per le elaborazioni che il chatbot effettuerà. La corretta struttura del file di configurazione è di grande importanza poiché definisce il processo di estrazione degli intenti e delle entità. Nelle Figure 4.16 e 4.17 viene riportato il file di configurazione utilizzato nel progetto.

```
! config.yml ×
! config.yml > ...
 9  pipeline:
10  # # No configuration for the NLU pipel
11  # # If you'd like to customize it, unco
12  # # See https://rasa.com/docs/rasa/tuni
13      - name: WhitespaceTokenizer
14      - name: RegexFeaturizer
15      - name: LexicalSyntacticFeaturizer
16      - name: CountVectorsFeaturizer
17        analyzer: word
18      - name: CountVectorsFeaturizer
19      - name: RegexEntityExtractor
20        analyzer: char_wb
21        min_ngram: 1
22        max_ngram: 4
23      - name: DIETClassifier
24        use_masked_language_model: True
25        epochs: 200
26        constrain_similarities: true
27      - name: EntitySynonymMapper
28      - name: ResponseSelector
29        epochs: 200
30        constrain_similarities: true
31      - name: FallbackClassifier
32        threshold: 0.7
33        ambiguity_threshold: 0.5
34
```

Figura 4.16: Pipeline di Rasa

Per comprendere la figura sopra riportata, verranno fornite spiegazioni dettagliate riguardo a tutti i parametri configurati per la pipeline dell'unità di comprensione dell' NLU utilizzata da Rasa.

- *WhitespaceTokenizer*: separa le parole dagli spazi bianchi.
- *RegexFeaturizer*: consente di identificare ed estrarre informazioni rilevanti dai testi, utilizzando espressioni regolari. Le espressioni regolari sono pattern di testo predefiniti che possono essere utilizzati per cercare corrispondenze o modelli specifici all'interno del testo.
- *LexicalSyntacticFeaturizer*: ha il compito di estrarre le caratteristiche lessicali e sintattiche dai testi di addestramento, per arricchire le informazioni utilizzate dal modello durante la fase di classificazione degli intenti e l'estrazione di entità.
- *CountVectorsFeaturizer*: converte il testo in una rappresentazione numerica basata sulla frequenza delle parole. Questa funzione conta il numero di volte che ciascuna parola appare in un determinato testo e crea un vettore di conteggio delle parole come caratteristica per l'addestramento del modello.
- *min/max_ngram*: indica la lunghezza minima o massima degli n-grammi considerati durante la tokenizzazione del testo. Gli n-grammi sono sequenze di n parole consecutive che vengono utilizzate per rappresentare il testo durante l'elaborazione del linguaggio naturale.

- *DIETClassifier*: è stato descritto nella Sezione 4.2.
- *EntitySynonymMapper*: è utilizzato per gestire i sinonimi delle entità. L'obiettivo di questo componente è quello di "mappare" le diverse varianti di un'entità in una forma standardizzata in modo da consentire una migliore comprensione del testo da parte del chatbot.
- *ResponseSelector*: è un componente utilizzato per selezionare una risposta predefinita da un insieme di risposte preconfigurate, in base all'intento dell'utente.
- *FallbackClassifier*: consente di gestire situazioni in cui il modello di elaborazione del linguaggio naturale non è in grado di riconoscere un'intent dell'utente o di estrarre entità dal messaggio. Viene utilizzato come meccanismo di fallback quando il modello principale non è sicuro delle proprie previsioni o non riesce a generare un'azione corrispondente.

L'ultima sezione del file di configurazione è dedicata alle policy (Figura 4.17). Queste ultime rappresentano le strategie o le regole utilizzate dal sistema di dialogo per prendere decisioni durante l'interazione con l'utente. Esse, definiscono come il chatbot deve selezionare l'azione successiva da eseguire, in base allo stato corrente della conversazione.

```
policies:
# # No configuration for policies was provided. The follo
# # If you'd like to customize them, uncomment and adjust
# # See https://rasa.com/docs/rasa/policies for more infor
  - name: MemoizationPolicy
  - name: RulePolicy
    core_fallback_threshold: 0.7
    core_fallback_action_name: "action_default_fallback"
    #enable_fallback_prediction: True
  - name: UnexpecTEDIntentPolicy
    max_history: 5
    epochs: 200
  - name: TEDPolicy
    max_history: 5
    epochs: 200
    constrain_similarities: true
```

Figura 4.17: Policy Rasa

In dettaglio:

- *MemorizationPolicy*: memorizza e recupera i dialoghi con l'utente al fine di migliorare l'efficienza e la coerenza delle risposte del chatbot. Quando viene rilevato uno stato di conversazione simile, la policy utilizza l'azione precedentemente eseguita come suggerimento per l'azione necessaria.
- *RulePolicy*: utilizza regole decise a priori dall'utente per definire il flusso di conversazione desiderato.
- *UnexpecTEDIntentPolicy*: gestisce gli intenti imprevisti o inaspettati durante l'interazione con il chatbot. Questa policy è progettata per affrontare situazioni in cui l'utente introduce un'intent che non è stato previsto o incluso nel training del modello. Invece di ignorare completamente l'intento inaspettato, *UnexpecTEDIntentPolicy* cerca di gestirlo in modo adeguato, ad esempio chiedendo all'utente di fornire ulteriori informazioni o di specificare l'intento desiderato.
- *TEDPolicy*: è una policy di apprendimento rinforzato (reinforcement learning) utilizzata per gestire le decisioni sull'interazione con l'utente all'interno del sistema di dialogo. Il

suo approccio è basato su reti neurali trasformative per l'estrazione delle funzionalità. Questa policy è in grado di gestire sia la classificazione degli intenti, che la previsione delle azioni da intraprendere durante la conversazione.

4.3.1 Progettazione degli intenti

In questa sezione, si analizzerà il contenuto del file nlu.yml, utilizzato nel contesto di questo progetto. Si dedicherà particolare attenzione all'illustrazione degli intenti e alla loro definizione, fornendo ulteriori dettagli e spiegazioni specifiche per alcuni di essi.

Gli intenti in Rasa seguono una struttura semplice e intuitiva. Come evidenziato dalla Figura 4.18 e dal primo intento creato (l'intento *saluto*), la struttura principale di un intento comprende il nome dell'intento stesso e i suoi esempi. È importante fornire esempi precisi e includerne almeno due. Infatti, maggiore è il numero di esempi forniti, migliore sarà la capacità del chatbot di addestrarsi correttamente e di prevedere l'intento dell'utente.

```yaml
1  version: "3.1"
2
3  nlu:
4  - intent: saluto
5    examples: |
6      - Hey
7      - Ciao
8      - Salve
9      - Buon giorno
10     - Buongiorno
11     - Buona sera
12     - Buonasera
13     - Buon pomeriggio
14     - Ciao caro
15     - Buongiorno, vorrei delle informazioni
16     - Salve, mi servirebbe dell'assistenza per
17     - che sai fare
18     - per caso sei in grado di fare altro
19     - vorrei altre informazioni
20     - Buongiorno
21     - We la
22     - Salve mi serve una mano a
```

Figura 4.18: Intento saluto

Nelle Figure 4.19 e 4.45 sono riportate le immagini che mostrano gli intenti creati.

```yaml
- intent: trattamento_stipendiale_e_pensionistico_previdenziale
  examples: |
    - procedura per il trattamento predivenziale
    - previdenziale
    - modulo trattamento previdenziale
    - trattamento previdenziale
    - Ho bisogno di cercare il modulo per richiedere una consulenza previdenziale.
    - Sto cercando il modulo per la richiesta di una consulenza previdenziale.
    - Sono alla ricerca del modulo per richiedere una consulenza sulla previdenza.
    - Vorrei trovare il modulo per la richiesta di una consulenza previdenziale.
    - Ho bisogno di aiuto per cercare il modulo per richiedere una consulenza previdenziale.
    - Sto cercando il modulo per la richiesta di una consulenza sulla previdenza pensionistica.
    - Sono alla ricerca del modulo per richiedere una consulenza previdenziale sulla pensione.
    - Vorrei trovare il modulo per la richiesta di una consulenza previdenziale riguardo la pensione.
    - Ho bisogno di cercare il modulo per richiedere una consulenza riguardo la previdenza pensionistica.
    - Sto cercando il modulo per la richiesta di una consulenza previdenziale per il trattamento pensionistico.
```

```yaml
- intent: trattamento_stipendiale_e_pensionistico_stipendiale
  examples: |
    - procedura per il trattamento stipendiale e pensionistico
    - come potrei inviare il modulo per il mio trattamento pensionistico
    - modulo per il trattamento pensionistico
    - qual'è il modulo per la consulenza stipendiale
    - qual'è il modulo per l'area consulenza stipendiale
    - vorrei scaricare il modulo per la consulenza stipendiale
    - non trovo il modulo per il trattamento pensionistico e stipendiale
    - modulo trattamento stipendiale
    - simulazione del calcolo del servizio utile per conoscere la data in cui si
    - matura il diritto a pensione di vecchiaia o anticipata e la data di
    - presentazione della relativa domanda di pensione
    - simulazione del calcolo del servizio utile per conoscere la data in cui si matura il diritto a pensione di vecchiaia
    - simulazione del calcolo dell ’importo della pensione al raggiungimento del requisito minimo pensionistico
    - simulazione del calcolo dell ’importo della pensione al raggiungimento
    - del requisito minimo pensionistico
    - simulazione del calcolo dell’ importo relativo al trattamento di fine servizio e dell’ indennita supplementare
    - simulazione del calcolo dell’ importo relativo al trattamento di fine
    - servizio e dell’ indennita supplementare
    - consulenza per la presentazione della domanda di riscatto di 1/5, costi e procedure
    - presentazione della domanda di riscatto di 1/5, costi e procedure
    - consulenza per la presentazione della domanda di prestito, cessione o delega
```

```yaml
- intent: cerca_segretario_provincia
  examples: |
    - cerco provincia
    - ho scelto la provincia


- intent: provincia
  examples: |
    - provincia
    - segretario della provincia
    - segretario di provincia


- intent: iscrizione
  examples: |
    - come posso iscrivermi
    - come effettuare l'iscrizione al sito
    - posso accedere al sito
    - come loggarmi nel sito
    - come posso far parte dell'associazione dei carabinieri
    - associazione dei carabinieri
    - vorrei efettuare l'iscrizione al sito
    - non so come ci si iscrive al sito
    - mi serve una mano ad iscrivermi al sito
    - vorrei una mano per l'iscrizione al sito
    - non riesco a trovare dove posso iscrivermi al sito
    - iscriverti al sito
```

```yaml
- intent: bot_challenge
  examples: |
    - Sei un bot?
    - Sei umano?
    - sto parlando con un bot?
    - sto parlando con un uomo?
    - sto parlando con una persona?
    - sto parlando con una persona umana?
    - sto parlando con una persona umana o un bot?
    - sto parlando con una persona umana o un robot?
    - sei un robot?
    - sei un uomo?

- intent: arrivederci
  examples: |
    - Ciao è stato un piacere darti assistenza
    - Spero di esserti stato utile
    - Arrivederci
    - Alla prossima
```

Figura 4.19: Intenti creati

```
- intent: reset_password
  examples: |
    - come posso fare il reset della password
    - ho dimenticato la password come potrei ripristinare la password
    - non mi ricordo la password potrei fare il recupero della password
    - password
    - non so dove ho lasciato la mia password
    - ho perso la mia password e non riesco ad accedere al sito
    - non riesco ad accedere al sito come devo fare?
    - problemi con la password
    - reset password
    - recuperare password
```

```
- intent: intent_ricerca_dei_documenti
  examples: |
    - vorrei cercare un modulo relativo al
    - trovami il documento del
    - mi serve il modulo del
    - documento riguardante il
    - cercami l'allegato che parla del
    - cerca il documento che inizia con
    - trova il documento
    - ceca il file
    - l'allegato
    - Dove posso trovare il documento sulla
    - Sono alla ricerca del documento di
    - Ho bisogno di un documento che
    - Qual è il nome del documento riguardante la
    - Qual è il nome del file riguardante la
    - Qual è il nome del documento riguardante il
    - Qual è il nome dell'allegato riguardante il
    - Qual è il nome del documento riguardante che
    - C'è un documento che descrive il ?
    - Sono alla ricerca di un documento sulle politiche di privacy.
```

```
- intent: poliza_legale_denuncia_sinistro
  examples: |
    - Devo attivare la polizza legale oppure devo fare la denuncia di un sinistro
    - polizza legale
    - denuncia di un sinistro
    - polizza
    - legale
    - sinistro
    - come attivare la polizza legale
    - fare la denuncia di un sinistro
    - denuncia sinistro
    - denuncia
```

```
- intent: trattamento_stipendiale_e_pensioninsisco_scelta
  examples: |
    - pensione
    - pensioni
    - stipendiale
    - stipendi
    - previdenziale
    - cna
    - tea
    - TEA
    - ter
    - TER
    - Quiescenza
    - attività economica
    - trattamento pensionistico
    - trattamento stipendiale e pensioninstico
    - moduli per il trattamento stipendiale e pensionistico
    - come accedere all'area per il trattamento pensioninstico e previdenziale
    - trova come accedere all'area per il trattamento pensioninstico e previdenziale
    - simulazione del calcolo dell' importo della pensione al raggiungimento del limite di eta
    - simulazione del calcolo dell' importo della pensione al raggiungimento
    - del limite di eta
    - del limite di età
    - limite di età
```

Figura 4.20: Intenti creati

Tra tutti gli intenti creati verrà fornita una spiegazione dettagliata di un tipo specifico di intento, che presenta una struttura leggermente diversa dagli altri. In particolare, per l'intento *intent_cerca_segretario_regione* viene utilizzata una lookup table (Figura 4.21).

```
- intent: intent_cerca_segretario_regione
  examples: |
    - cercare il segretario di regione
    - trovare il segretario regionale
    - trova segretario
    - trova il segretario
    - cerca segretario
    - segretario
    - segretario della regione
    - chi è il segretario regionale
    - qual'è il segretario di regione
    - qual'è il segretario della regione [molise](regione)
    - vorrei cercare il segretario della regione
    - vorerei cercare il segretario della redione [calabria](regione)
    - il segretario di regione [marche](regione)
    - vorrei sapere qual'è il segretario della regione [emilia](regione)
    - cerca il segretario dell'[lazio](regione)
    - cerca il segretario della regione
    - cerca il segretario della regione [lombardia](regione)
    - chi è il segretario regionale [abruzzo](regione)
    - [abruzzo](regione)
    - [basilicata](regione)
```

Figura 4.21: Intento Cerca Segretario Regione

Le lookup table consentono di associare una lista predefinita di valori a un determinato intento. Questa lista di valori viene utilizzata come riferimento per l'individuazione dell'intento durante il processo di comprensione del linguaggio naturale. Quando un'annotazione di testo coincide con uno dei valori presenti nella lookup table, viene assegnato automaticamente l'intento corrispondente. La lookup table è utile quando si desidera riconoscere pattern specifici o parole chiave in modo rapido ed efficiente.

Per associare un'intento a una lookup table, come mostrato nella Figura 4.21, si inserisce il valore da confrontare tra le parentesi quadre e il nome della lookup table tra le parentesi tonde. Inoltre, per definire la lookup table, è stato creato un file apposito con lo stesso nome della lookup (Figura 4.22). Questo passaggio è fondamentale perché durante il processo di

comprensione del linguaggio naturale (NLU), il chatbot può individuare correttamente la lookup table corrispondente.

```
nlu:

- lookup: regione
  examples: |
    - abruzzo
    - Abruzzo
    - basilicata
    - Basilicata
    - calabria
    - Calabria
    - campania
    - Campania
    - emilia-romagna
    - Emilia-Romagna
    - emilia
    - Emilia
    - emilia romagna
    - Emilia Romagna
    - friuli-venezia giulia
    - Friuli-venezia Giulia
```

Figura 4.22: Lookup Table

4.3.2 Progettazione delle storie

Le storie in Rasa rivestono un ruolo fondamentale nella creazione di interazioni dinamiche tra il chatbot e l'utente, rendendo l'esperienza simile a un'interazione con una persona reale. La struttura delle storie (Figura 4.23), simile a quella degli intenti, è caratterizzata da una semplice composizione che include:

- *story*: nome della storia;
- *steps*: istruzione che raccoglie tutti gli eventi e le azioni che si verificano durante un'interazione;
- *intent*: intento che attiva la storia;
- *action*: azione che viene eseguita dopo che il chatbot ha compreso l'intento dell'utente.

```
stories:
- story: saluto
  steps:
  - intent: saluto
  - action: action_secretary
```

Figura 4.23: Esempio di una storia

La struttura descritta precedentemente può essere considerata come la base per la creazione delle storie. Le storie possono essere composte da più intenti e azioni, seguendo la regola che dopo l'intento deve seguire necessariamente un'azione. Tuttavia, questa regola non si applica alle azioni, che possono essere seguite da altri intenti o azioni. Oltre agli intenti e alle azioni, è possibile includere condizioni che guidino il chatbot nella comprensione della storia da seguire durante l'interazione con l'utente.

Durante la creazione del chatbot, tra le molteplici condizioni disponibili, è stata utilizzata una in particolare, chiamata "checkpoint". Un checkpoint è un meccanismo che consente di salvare lo stato di avanzamento di una conversazione. Viene utilizzato per controllare se un determinato punto è stato raggiunto durante l'interazione con l'utente. Esso è utile per gestire la logica del flusso di conversazione, consentendo al chatbot di prendere decisioni basate sulle azioni precedenti o sullo stato attuale della conversazione. I checkpoint vengono definiti nelle storie come delle condizioni che devono essere soddisfatte affinché l'interazione possa procedere secondo un percorso specifico.

Il checkpoint, nel progetto, è stato utilizzato per gestire in modo specifico il trattamento stipendiale e pensionistico dell'utente (Figura 4.24).

```
- story: trattamento_stipendiale_e_pensionistico_gestione
  steps:
  - intent: trattamento_stipendiale_e_pensioninsisco_scelta
  - action: utter_scelta_trattamento_pensioninstico
  - checkpoint: checkpoint_stipendiale_pensionistico

- story: trattattamento stipendiale
  steps:
  - checkpoint: checkpoint_stipendiale_pensionistico
  - intent: trattamento_stipendiale_e_pensionistico_stipendiale
  - action: utter_stipendiale

- story: trattattamento previdenziale
  steps:
  - checkpoint: checkpoint_stipendiale_pensionistico
  - intent: trattamento_stipendiale_e_pensionistico_previdenziale
  - action: utter_previdenziale
```

Figura 4.24: Utilizzo del checkpoint

Nella Figura 4.25 sono illustrate tutte le storie sviluppate per implementare il chatbot. Non si entrerà nel dettaglio poiché la maggior parte delle storie è composta da intenti e azioni senza condizioni particolari.

```
stories:
- story: saluto
  steps:
  - intent: saluto
  - action: action_secretary

- story: bot challenge dopo saluto
  steps:
  - intent: bot_challenge
  - action: utter_iamnotabot

- story: iscrizione
  steps:
  - intent: iscrizione
  - action: utter_iscrizione

- story: reset password
  steps:
  - intent: reset_password
  - action: utter_reset_password

- story: arrivederci
  steps:
  - intent: arrivederci
  - action: utter_arrivederci

- story: opzione segretario regione
  steps:
  - intent: intent_cerca_segretario_regione
  - action: utter_segretario
  - action: action_find_region2
```

```
- story: polizza legale o denuncia sinistri
  steps:
  - intent: poliza_legale_denuncia_sinistro
  - action: action_polizza_o_denuncia
```

```
- story: trattamento_stipendiale_e_pensionistico_gestione
  steps:
  - intent: trattamento_stipendiale_e_pensioninsisco_scelta
  - action: utter_scelta_trattamento_pensioninstico
  - checkpoint: checkpoint_stipendiale_pensionistico

- story: trattattamento stipendiale
  steps:
  - checkpoint: checkpoint_stipendiale_pensionistico
  - intent: trattamento_stipendiale_e_pensionistico_stipendiale
  - action: utter_stipendiale

- story: trattattamento previdenziale
  steps:
  - checkpoint: checkpoint_stipendiale_pensionistico
  - intent: trattamento_stipendiale_e_pensionistico_previdenziale
  - action: utter_previdenziale

- story: cerca trattattamento stipendiale
  steps:
  - intent: trattamento_stipendiale_e_pensionistico_stipendiale
  - action: utter_stipendiale

- story: cerca atrattattamento previdenziale
  steps:
  - intent: trattamento_stipendiale_e_pensionistico_previdenziale
  - action: utter_previdenziale

- story: ricercafile
  steps:
  - intent: intent_ricerca_dei_documenti
  - action: action_search_parola
```

Figura 4.25: Storie

4.3.3 Progettazione delle regole

In questa sezione, saranno descritte le regole; queste rappresentano storie più strutturate, in cui il chatbot segue le istruzioni e le condizioni imposte. La differenza principale tra una storia e una regola è che una regola deve completare tutti i passaggi assegnati, mentre una storia può interrompersi.

Durante l'esecuzione di una storia, il chatbot può passare tra diverse storie, in base alle risposte dell'utente, senza necessariamente concludere la storia in corso.

Nel caso delle regole, invece, prima di passare ad un'altra regola o storia, la regola in esecuzione deve completare tutti i suoi passaggi.

Anche per la creazione di una regola la struttura è molto simile a quella delle storie (Figura 4.26).

```
- rule: dopo il saluto
  steps:
  - intent: saluto
  - action: action_secretary
```

Figura 4.26: Esempio di una regola

Così come per le storie, le regole possono essere condizionate in modo da essere attivate ed eseguite solo se determinate condizioni sono soddisfatte. Nel progetto in questione, sono state utilizzate due condizioni specifiche: *conversation_start* e *slot_was_set*.

Entrambe le condizioni sono valutazioni booleane che restituiscono un valore vero o falso. La prima condizione specifica che la regola viene attivata solo quando la conversazione tra il chatbot e l'utente è appena iniziata, cioè al primo avvio del chatbot (Figura 4.27).

```
rules:

- rule: saluto iniziale
  conversation_start: true
  steps:
  - intent: saluto
  - action: action_secretary
```

Figura 4.27: Condizione *conversation_start*

La seconda condizione, invece, viene attivata solo quando uno o più slot hanno un valore assegnato. Nella Figura 4.28 è evidenziato che la regola viene attivata solo quando lo slot *regione* ha un valore.

```
- rule: per cercare la regione
  condition:
    - slot_was_set:
      - regione: true
  steps:
  - intent: intent_cerca_segretario_regione
  - action: action_find_region
```

Figura 4.28: Condizione *slot_was_set*

Nella Figura 4.29 sono illustrate tutte le regole sviluppate per implementare il chatbot. Non si entretà nel dettaglio, poiché la maggior parte delle regole è composta da intenti e azioni, senza condizioni particolari.

```
rules:

- rule: saluto iniziale
  conversation_start: true
  steps:
  - intent: saluto
  - action: action_secretary
```

```
- rule: saluto iniziale telegram
  conversation_start: true
  steps:
  - intent: start
  - action: action_secretary

- rule: dopo saluto iniziale telegram
  steps:
  - intent: start
  - action: action_secretary

- rule: dopo il saluto
  steps:
  - intent: saluto
  - action: action_secretary


- rule: fallimento
  steps:
  - intent: nlu_fallback
  - action: action_default_fallback
```

```
- rule: per cercare la regione
  condition:
    - slot_was_set:
      - regione: true
  steps:
  - intent: intent_cerca_segretario_regione
  - action: action_find_region


- rule: scegli la provincia
  steps:
  - intent: cerca_segretario_provincia
  - action: action_find_choice_province

- rule: provincia
  steps:
  - intent: provincia
  - action: action_find_province
```

Figura 4.29: Regole

Infine, anche se le regole possono sembrare più convenienti rispetto alle storie durante la fase di progettazione, a causa della loro rigidità nell'eseguire tutti i passaggi assegnati, è consigliabile evitarne un uso eccessivo poiché potrebbe compromettere la naturalezza del chatbot. Infatti, progettare un chatbot basato principalmente su regole impedirebbe di conferire ad esso la dinamicità necessaria per renderlo più "umano", trasformandolo in un semplice robot che esegue istruzioni specifiche.

4.4 Progettazione delle risposte e delle azioni del chatbot

Dopo aver descritto la costruzione degli intenti, delle storie e delle regole, sarà ora illustrato il processo di progettazione delle risposte e delle azioni. Prima di approfondire questi aspetti, è fondamentale introdurre il file *domain.yml*.

Questo è un file di configurazione utilizzato per definire il dominio del chatbot e contiene informazioni cruciali, come gli intenti, le azioni, le entità, i modelli di linguaggio, le risposte in formato testuale, i form e altre impostazioni di configurazione.

La corretta configurazione del file *domain.yml* è essenziale per definire il dominio del chatbot e guidarne il comportamento durante una conversazione. Questa sezione del capitolo introduce il file perché contiene gli elementi fondamentali per la progettazione delle risposte e delle azioni.

Di seguito verrano riportati dei frammenti di esso.

Come si può osservare dalla Figura 4.30, nel file vengono riportati in elenco tutte le entità, gli intenti e le azioni che il chatbot deve gestire. È di fondamentale importanza creare questi elenchi poiché se, ad esempio, un intento non è presente nell'elenco, anche se è stato definito nel file *intent.yml*, il chatbot non ne sarà a conoscenza e, di conseguenza, le storie e le regole non potranno essere eseguite. Si può affermare che la creazione di questi elenchi consente di "mappare" la conoscenza del chatbot.

Per quanto concerne le entità e gli slot, è essenziale definire correttamente il metodo di mappatura dei loro valori.

```
intents:
  - saluto
  - start
  - bot_challenge
  - arrivederci
  - intent_cerca_segretario_regione
  - intent_regioni
  - provincia
  - iscrizione
  - reset_password
  - cerca_segretario_provincia
  - trattamento_stipendiale_e_pensionistico_stipendiale
  - trattamento_stipendiale_e_pensionistico_previdenziale
  - trattamento_stipendiale_e_pensioninsisco_scelta
  - intent_ricercafile
  - intent_ricerca_dei_documenti
  - poliza_legale_denuncia_sinistro
```

```
entities:
  - regione
  - province
  - choice_province
  - slot_ricerca_dei_documenti
  - slot_ricerca_regione
  - documenti
```

```
actions:
- action_secretary
- action_find_region
- action_find_region2
- action_find_province
- action_find_choice_province
- action_default_fallback
- action_search_parola
- action_polizza_o_denuncia
```

Figura 4.30: Intenti, Entity, e Action del file *domain*

Come evidenziato nella Figura 4.31, sono presenti due metodologie per ottenere il valore da assegnare.

```
slot_ricerca_regione:
  type: text
  influence_conversation: true
  mappings:
    - type: from_text
      intent: intent_regioni
```

```
documenti:
  type: text
  influence_conversation: true
  mappings:
    - type: from_entity
      entity: documenti
```

Figura 4.31: Mappatura degli slot e delle entity

In entrambe le immagini, si evince la stessa struttura di base composta da:

- *type*
- *influence_conversation*
- *mappings*
  - *type*
  - *intent\entity*

Dopo aver specificato il nome dello slot o dell'entità, il passo successivo consiste nella sua dichiarazione, in cui si stabilisce il tipo di dato dell'elemento; che può essere testuale, numerico, booleano, etc.

Successivamente è possibile impostare alcuni parametri per attribuire maggiore importanza allo slot o all'entità durante la conversione. Ad esempio, utilizzando l'istruzione *influence_conversation:true*, viene assegnato un peso specifico allo slot o all'entità quando uno di essi acquisisce un valore. Questo peso aiuta il chatbot a comprendere meglio l'intento che si desidera individuare.

L'istruzione successiva, chiamata *mappings*, specifica quando il chatbot deve riconoscere l'informazione fornita in input e trasformarla nell'informazione da aggiungere allo slot o all'entità. Nelle due casistiche illustrate nell'immagine sono evidenziate le differenze; vengono anche fornite le istruzioni per il mapping di un'entità e di uno slot. Nella parte destra dell'immagine, le istruzioni *type: from_entity* e *entity: documenti* indicano che il chatbot assegnerà il valore qualora riconosca l'entità "documenti" nell'input dell'utente. Mentre, nella parte sinistra, con le istruzioni *type: from_text* e *intent: intent_regioni*, il chatbot assegnerà lo slot considerando l'input dell'utente in corrispondenza dell'intento *intent_regioni*.

Dopo aver introdotto alcuni concetti relativi al file *domain*, si procede a descrivere l'ultima sezione di questo file e a introdurre le risposte e le azioni del chatbot. La sezione finale del file *domain* è dedicata alle *responses* (risposte). In questa sezione è possibile inserire principalmente risposte di tipo testuale, ma è anche possibile includere risposte contenenti immagini o pulsanti interattivi. Le risposte del chatbot sono identificate dal prefisso *utter_* seguito dal nome della risposta. Inoltre, per ciascuna risposta è possibile assegnare più varianti, come illustrato nella Figura 4.32.

```
responses:
  utter_saluto:
    - text: Ciao, come posso aiutarti?
    - text: Ciao caro, come posso esserti utile?
```

Figura 4.32: Risposte del Chatbot

Nella Figura 4.33 viene illustrato come creare una risposta che includa due pulsanti; in base al pulsante selezionato dall'utente, il chatbot indirizzerà la conversazione a un intento specifico, indicato nell'istruzione "payload".

```
utter_scelta_trattamento_pensioninstico:
  - text: "Dimmi quale area del trattamento pensionistico preferisci"
    buttons:
    - title: "Area Consulenza Stipendiale"
      payload: "/trattamento_stipendiale_e_pensionistico_stipendiale"
    - title: "Area Consulenza Previdenziale"
      payload: "/trattamento_stipendiale_e_pensionistico_previdenziale"
```

Figura 4.33: Risposte del chatbot con pulsanti interattivi

Gli *utter* sono considerati risposte che, a tutti gli effetti, rappresentano delle azioni. Come riportato nella Figura 4.25, esse vengono utilizzate all'interno delle storie come azioni e, di conseguenza, forniscono una risposta all'utente, in relazione a un determinato intento.

Grazie agli *utter*, è stato possibile completare con facilità alcuni task assegnati dal Sindacato Italiano Militari Carabinieri. Attraverso queste risposte, sono state soddisfatte le seguenti richieste:

- l'iscrizione al sito (Figura 4.34);
- il recupero della Password (Figura 4.35);

- la procedura per il trattamento Pensionistico e il trattamento Stipendiale (Figure 4.36 e 4.37)

```
utter_iscrizione:
- text: "Gentile collega ,
        in questa pagina puoi effettuare la procedura d'iscrizione online e i necessari documenti verranno automati

        Per completare l'iscrizione è poi necessario, una volta che avrai firmato la delega, che tu faccia l'Upload
        Scarica delega[website](https://www.simcarabinieri.com/docs/delega_orig.pdf) \n

        L'iscrizione ha validità dal 1° giorno del mese successivo all'iscrizione fino al 31 dicembre di ogni anno

        Grazie per la fiducia accordataci."
```

Figura 4.34: Utter relativo all'iscrizione al sito

```
utter_reset_password:
- text: " Per ripristinare la password andare sulla propria area riservata.
        I passi da eseguire sono:\n

        1) Andare nell'Area utenti (https://simcarabinieri.com/login.php)\n

        2) Cliccare sull link password dimenticata \n

        3) Successivamente dovrai inserire la tua mail e cliccare il tasto Invia Richiesta\n

        4) Controllare sulla casella di posta [inserita nel punto 3] l'arrivo di una nuova comunicazione.

        5) Aprire la mail e cliccare sul link per il recupero della password\n

        6) Inserire la nuova password"
```

Figura 4.35: Utter relativo al reset della password

```
utter_previdenziale:
  - text: "Ecco il modulo da scaricare\n https://www.simcarabinieri.com/uploads/dipa
          \n
           Successivamente nella pagina https://www.simcarabinieri.com/dipartimen
           \n
           Allegare la documentazione:\n
           a) Modulo richiesta consulenza previdenziale\n
           b) Estratto conto contributivo INPS (esclusivamente in formato .xml)\n
           c) Ultimo statino stipendiale\n
           E premere il tasto Invia"
```

Figura 4.36: Utter relativo al trattamento previdenziale

```
utter_stipendiale:
  - text: "Ecco il modulo da scaricare\n https://www.simcarabinieri.com/uploads/dipar
           \n
           Successivamente nella pagina https://www.simcarabinieri.com/dipartimento
           \n
           Allegare la documentazione:\n
           a) Modulo richiesta consulenza stipendiale\n
           b) Ultimo statino stipendiale\n
           E premere il tasto Invia"
```

Figura 4.37: Utter relativo al trattamento stipendiale

Per quanto riguarda i due trattamenti (Figure 4.36 e 4.37), è stato creato un tipo di risposta aggiuntiva, nel caso in cui il chatbot non sia in grado di distinguere correttamente quale dei due trattamenti sia stato richiesto dall'input dell'utente, considerando che le due richieste hanno molte parole in comune. In situazioni di incertezza, è stata creata una risposta con l'aggiunta di due pulsanti, i quali consentono all'utente di interagire. Cliccando su uno dei due pulsanti, l'utente può indirizzare il chatbot verso l'intento desiderato. Attraverso l'utilizzo della funzione *playload*, come evidenziato nella Figura 4.38, il chatbot viene guidato verso l'intento corretto.

```
utter_scelta_trattamento_pensioninstico:
  - text: "Dimmi quale area del trattamento pensionistico preferisci"
    buttons:
    - title: "Area Consulenza Stipendiale"
      payload: "/trattamento_stipendiale_e_pensionistico_stipendiale"
    - title: "Area Consulenza Previdenziale"
      payload: "/trattamento_stipendiale_e_pensionistico_previdenziale"
```

Figura 4.38: Utter relativo alla scelta del trattamento

Tuttavia, gli *utter* possono presentare alcune limitazioni quando si desidera fornire risposte specifiche o calcolate. Per affrontare questa situazione, Rasa offre la possibilità di generare risposte personalizzate mediante l'uso del codice Python.

Sfruttando la flessibilità del linguaggio Python e le API di Rasa, si è definito manualmente tutte le azioni ancora mancanti.

La prima azione personalizzata che è stata sviluppata è quella relativa al messaggio iniziale del chatbot dopo l'avvio. Una volta avviata la conversazione, il chatbot inizia presentandosi come "BotCommander" ed elencando le funzioni che può svolgere. Per ottenere questo risultato è stato creato un file Python dedicato, in cui è stata definita una classe che estende la classe Action (Figura 4.39).

La Figura 4.39 mostra che l'azione personalizzata è composta da due funzioni: *name* e *run*. Nella funzione *name* viene assegnato un nome all'azione personalizzata, mentre nella funzione *run* vengono inserite le istruzioni che l'azione deve eseguire. Per semplificare e organizzare il codice, è stata creata una nuova classe chiamata *Utils* che contiene tutte le funzioni necessarie a completare i task richiesti dal SIM. La classe *Utils* sarà utilizzata dalle diverse azioni personalizzate, che saranno spiegate successivamente.

Nell'azione *action_secretary* (Figura 4.39) viene costruito un vettore di stringhe, contenente il saluto e le istruzioni che il chatbot può eseguire all'interno della classe *Utils*. Successivamente, nella funzione *run*, queste stringhe vengono stampate utilizzando il dispatcher, una funzione di Rasa che consente di visualizzarle.

```
class SecretaryInizioAction(Action):

    def name(self) -> Text:
        return "action_secretary"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            response_builder = SecretaryInizioAction.Utils()
            response=response_builder.build_response()
            for text in response:
                dispatcher.utter_message(text)

            return []

    class Utils:

        def build_response(self):
            messaggio_iniziale=[]
            messaggio_iniziale_1="Ciao sono il tuo BotComander,e sono qui per servirti nel miglior modo possibile.\n"
            messaggio_iniziale.append(messaggio_iniziale_1)
            messaggio_iniziale_2="Per ora sono solo addestrato per poterti guidara a:\n"
            messaggio_iniziale.append(messaggio_iniziale_2)
            messaggio_iniziale_3="-Recuperare la password\n"
            messaggio_iniziale.append(messaggio_iniziale_3)
            messaggio_iniziale_4="-Iscriverti al sito\n"
            messaggio_iniziale.append(messaggio_iniziale_4)
            messaggio_iniziale_5="-Cercare il Segretario di Regione"
            messaggio_iniziale.append(messaggio_iniziale_5)
            messaggio_iniziale_6="-Procedura per il Trattamento Stipendiale e Pensionistico"
            messaggio_iniziale.append(messaggio_iniziale_6)
            messaggio_iniziale_7="-Cercare un allegato"
            messaggio_iniziale.append(messaggio_iniziale_7)
            messaggio_iniziale_8="-Gestione della polizza e dei sinistri"
            messaggio_iniziale.append(messaggio_iniziale_8)
            return messaggio_iniziale
```

Figura 4.39: *SecretaryInizioAction*

In modo simile all'azione precedente, anche per la gestione della polizza e dei sinistri è stata adottata la stessa struttura. Tuttavia, a differenza dell'azione precedente, sono state inserite immagini tra la stampa delle diverse stringhe nel corso dell'esecuzione dell'azione (Figure 4.40 e 4.41).

```
class SecretaryPolizzaAction(Action):

    EasyCode: Explain
    def name(self) -> Text:
        return "action_polizza_o_denuncia"

    EasyCode: Explain
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            response_builder = SecretaryPolizzaAction.Utils()
            image_path = "actions/utils/databases/image/polizza_o_sinistri/"
            response=response_builder.build_response()
            for index,text in enumerate(response):
                dispatcher.utter_message(text)
                if index<5:
                    immagine=image_path+f"{index+1}.png"
                    dispatcher.utter_image_url(image=immagine)
            return []
```

Figura 4.40: *SecretaryPolizzaAction-1*

```
class Utils:

    EasyCode: Explain
    def build_response(self):
        messaggio_iniziale=[]
        messaggio_iniziale_1="Step 1: Accedi al sito.\n"
        messaggio_iniziale.append(messaggio_iniziale_1)
        messaggio_iniziale_2="step 2: Entra nella tua area personale e clicca su tutela legale situato sotto all'immagine della
        messaggio_iniziale.append(messaggio_iniziale_2)
        messaggio_iniziale_3="Step 3: A questo punto si aprirà un sito. In alto clicca sul pulsante denuncia un sinistro\n"
        messaggio_iniziale.append(messaggio_iniziale_3)
        messaggio_iniziale_4="Step 4: Scarica il modulo cliccando sul pulsante Download oppure cliccando qui sotto.\n"
        messaggio_iniziale.append(messaggio_iniziale_4)
        messaggio_iniziale_5="https://www.simcarabinieri.com/tutela-legale/download/modello-denuncia-sinistro.pdf"
        messaggio_iniziale.append(messaggio_iniziale_5)
        messaggio_iniziale_6="Step 5: Successivamente compila il modulo e poi invialo alle email:"
        messaggio_iniziale.append(messaggio_iniziale_6)
        messaggio_iniziale_7="tutelalegale@simcarabinieri.cc "
        messaggio_iniziale.append(messaggio_iniziale_7)
        messaggio_iniziale_8="simcarabinieri@afifamilyonline.it"
        messaggio_iniziale.append(messaggio_iniziale_8)
        return messaggio_iniziale
```

Figura 4.41: *SecretaryPolizzaAction-2*

Durante la creazione del chatbot, sono stati inclusi intenti e azioni aggiuntivi, al di là delle specifiche fornite dal SIM, al fine di rendere il chatbot più umano. Tuttavia, non ci si soffermerà su queste specifiche, poiché si tratta di intenti e azioni semplici, come *arrivederci*, *se sei un bot*, etc.

4.4.1 Fallback Action

Durante una conversazione con un utente, il chatbot cerca di prevedere e interpretare l'intento dell'utente in base ai suoi input. Tuttavia, poiché gli utenti possono fornire una vasta gamma di input, può accadere che il chatbot non riesca a comprendere completamente l'intento desiderato e risponda con l'intento più probabile, calcolato tramite DIETClassifier. Al fine di evitare risposte inappropriate, è stato implementato un meccanismo di *fallback_action*. La *fallback_action* viene attivata quando il grado di confidenza associato ad un intento predetto dal chatbot non supera una determinata soglia. Questa soglia è specificata nel file di configurazione *config.yml* (Figura 4.42). L'uso della *fallback_action* richiede l'inclusione di un FallBackClassifier nella pipeline del chatbot (Figura 4.43).

```
- name: RulePolicy
  core_fallback_threshold: 0.7
  core_fallback_action_name: "action_default_fallback"
```

Figura 4.42: Core *FallBack-config.yml*

```
- name: FallbackClassifier
  threshold: 0.7
  ambiguity_threshold: 0.5
```

Figura 4.43: Fallback *Classifier-config.yml*

Quando il FallBackClassifier determina che la confidenza associata all'intento predetto è inferiore a una soglia predefinita (nel caso specifico, 0.7), viene attivata un'azione di fallback.

L'azione costruita consiste nell'indirizzare l'utente a interagire con il chatbot utilizzando pulsanti al fine di guidarlo verso l'intento desiderato, in base all'input precedente. Dopo un breve messaggio introduttivo che afferma: *Attualmente non sono in grado di gestire questa richiesta. Ricominciamo da capo. Ti darò una mano*, il chatbot elenca nuovamente tutte le operazioni disponibili, come già fatto durante il suo primo avvio. Tuttavia, questa volta vengono elencate sotto forma di pulsanti, consentendo all'utente di selezionare direttamente l'intento desiderato con un semplice clic.

I suddetti pulsanti vengono creati nella classe *Utils* (Figura 4.45), mentre, tramite la funzione *run*, viene stampato il messaggio di errore in aggiunta ai pulsanti che consentono all'utente di interagire (Figura 4.44).

```
class fallbackfindAction(Action):

    def name(self) -> Text:
        EasyCode: Explain
        return "action_default_fallback"

    def run(
        EasyCode: Explain
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[Dict[Text, Any]]:

        dispatcher.utter_message(f"Attualmente non riesco a gestire questa richiesta. \n"
                                +"Rincominciamo da capo. Ti do una mano")

        response_builder=fallbackfindAction.Utils()
        dispatcher.utter_message(text = response_builder.build_response(), buttons = response_builder.get_buttons())

        return [Restarted()]
```

Figura 4.44: Action *Fallback-1*

```
class Utils:

    def build_response(self):
        EasyCode: Explain
        return f"Scegli una delle seguenti opzioni oppure chiedimi qualcosa"

    def get_buttons(self) -> List[Dict[Text, Any]]:
        EasyCode: Explain
        return [
            {"title": "Iscrivimi", "payload": "/iscrizione"},
            {"title": "Reset Password", "payload": "/reset_password"},
            {"title": "Cerca Segretario", "payload": "/intent_cerca_segretario_regione"},
            {"title": "Procedura per il Trattamento Stipendiale e Pensionistico", "payload": "/trattamento_stipendiale_e_pensio
            {"title": "Cercare un allegato", "payload": "/intent_ricerca_dei_documenti"},
            {"title": "Gestione della polizza e dei sinistri", "payload": "/poliza_legale_denuncia_sinistro"}
        ]
```

Figura 4.45: Action *Fallback-2*

4.4.2 Realizzazione della funzione "ricerca segretario"

La ricerca del segretario di regione ha richiesto un lavoro più complesso e accurato rispetto alle altre azioni. A differenza di esse, infatti questa azione richiedeva un'elaborazione più sofisticata dell'input fornito. Infatti, il chatbot doveva essere in grado di comprendere se l'intento fosse la ricerca del segretario insieme al nome della regione, o solamente il nome della regione. Durante la fase di progettazione, è stato studiato un approccio tale per cui qualsiasi elemento inserito come input potesse essere collegato alla ricerca del segretario. A tal fine, è stata creata un'entità denominata *regione* che, successivamente, è stata inclusa negli esempi dell'intento per addestrare il modello.

L'entità *regione* è stata introdotta per garantire che qualsiasi menzione di una regione nel testo, indipendentemente dal contesto, fosse associata alla ricerca del segretario. Il chatbot, riconoscendo l'intento e la regione, acquisce il valore della regione fornita in input e lo elabora nell'azione personalizzata chiamata *action_find_region* (Figura 4.46). In questa azione, la prima cosa che viene fatta è caricare tutte le regioni presenti in un file, in una variabile di tipo array. Successivamente, si estrapola il valore dell'entità utilizzando il tracker, un componente di Rasa che traccia lo stato della conversazione e memorizza i valori delle entità o degli slot. Una volta ottenuto il valore, si verifica se l'entità è presente e se corrisponde a una delle

regioni nell'array. Nel caso in cui non fosse stata trovata una corrispondenza per entrambe le opzioni, si richiede all'utente di fornire ulteriori dettagli per aiutare il chatbot a comprendere meglio la risposta.

```
class SecretaryfindAction(Action):

    knwolodge=Path("data/regione.txt").read_text().split("\n")

    EasyCode: Explain
    def name(self) -> Text:
        return "action_find_region"

    EasyCode: Explain
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            if tracker.latest_message['entities']:

                for text in tracker.latest_message['entities']:

                    if text['entity']=='regione' and text['extractor']=='RegexEntityExtractor':
                        name=text['value'].lower()
                        if name in self.knwolodge:
                            response_builder = SecretaryfindAction.Utils()
                            chech_exsist=response_builder.build_response(tracker,name)
                            if chech_exsist==[]:
                                dispatcher.utter_message(text = "Non è stato ancora nominato il Segretario")
                            else:
                                dispatcher.utter_message(text = "\n".join(map(str, response_builder.build_response(tracker,name)))

                        else:
                            dispatcher.utter_message(text=f"Hai scritto male {name}")
            else:
                dispatcher.utter_message(text=f"Hai scritto {tracker.latest_message['text']}. Forse intendevi un'altra regione?")



            return [UserUtteranceReverted()]
```

Figura 4.46: Action *Cerca Segretario di Regione-1*

Se la regione specificata dall'utente corrisponde con successo a una delle regioni nell'array, il chatbot fornirà le informazioni relative al segretario di regione. Per l'estrapolazione di tali informazioni, è stata utilizzata la classe *Utils*, che contiene due funzioni. Nella funzione *build_response* (Figura 4.47), viene costruita la risposta, acquisendo i dati del segretario di regione da un file JSON. Per il recupero dei dati dal file, è stato adottato il pattern DAO (Data Access Object). Tale pattern separa la logica di accesso ai dati dalla logica di business all'interno di un'applicazione. Il suo obiettivo principale è fornire un'interfaccia uniforme per l'accesso ai dati, indipendentemente dalla fonte di questi ultimi (ad esempio database, file, servizi esterni, etc.). Nel progetto, è stato creato un DAO appositamente collocato in una cartella dedicata. Una volta ottenuti i dati, questi vengono inseriti in un vettore e, successivamente, restituiti al dispatcher, che si occupa della loro stampa nel video.

```
class Utils:

    EasyCode: Explain
    def build_response(self, tracker: Tracker, name_region:str):
        take_segretary=secretary.instance().readSegretaryregion(name_region)
        reply=[]
        print(take_segretary)
        if take_segretary:

            for index in take_segretary:

                reply.append(index)
        else:
            reply=[]

        return reply

    EasyCode: Explain
    def get_buttons(self,my_region) -> List[Dict[Text, Any]]:
        info=my_region
        print(info)
        return [
            {"title": "Cerca per Provincia", "payload": '/cerca_segretario_provincia{"choice_province":"'+info+'" }'},
            {"title": "Arrivederci", "payload": "/arrivederci"}
        ]
```

Figura 4.47: Action *Cerca Segretario di Regione-2*

La seconda funzione, chiamata *get_button* (Figura 4.47), consente al chatbot di restituire una serie di pulsanti. Nonostante non sia stata richiesto dal SIM, si è deciso di offrire agli utenti la possibilità di cercare il segretario della propria provincia. A tal fine, vengono creati due pulsanti che consentono di effettuare la ricerca del segretario provinciale, basandosi sulla regione appena cercata, oppure di terminare la ricerca con un saluto di *arrivederci*. Se l'utente desidera cercare il segretario di provincia, e quindi seleziona il pulsante *cerca segretario di provincia*, il chatbot eseguirà un'altra azione chiamata *action_find_choice_province* (Figura 4.48).

```
class SecretaryfindchoiceAction(Action):

    EasyCode: Explain
    def name(self) -> Text:
        return "action_find_choice_province"

    EasyCode: Explain
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            slot_province=tracker.get_slot("choice_province")
            response_builder = SecretaryfindchoiceAction.Utils()
            print(slot_province)
            dispatcher.utter_message(text = response_builder.build_response(tracker,slot_province),buttons = response_builder.get_b

            return []

    class Utils:

        EasyCode: Explain
        def build_response(self, tracker: Tracker, name:str):
            return f"Le provincie della regione {name} sono:"

        EasyCode: Explain
        def get_buttons(self,my_region) -> List[Dict[Text, Any]]:
            take_province=secretary.instance().readlistprovince(my_region)
            list_button=[]
            for index in list(take_province.columns):
                info=str(my_region+"-"+index)
                list_button.append({"title": f"{index}", "payload": '/provincia{"province":"'+info+'" }'})
            return list_button
```

Figura 4.48: Action per la scelta della provincia

In questa azione, utilizzando la regione trovata precedentemente, viene richiesto all'utente di selezionare la provincia desiderata premendo su uno dei pulsanti che rappresentano le diverse province disponibili.

Quando l'utente seleziona una provincia, viene attivata un'altra azione chiamata *action_find_province* (Figura 4.49) che si occupa di restituire i dati del segretario di provincia. In questa azione, utilizzando la classe *Utils*, nella funzione "get_info", vengono ottenuti i dati del segretario di provincia basandosi sulla provincia passata come slot attraverso il pulsante della precedente azione. I dati vengono successivamente forniti al dispatcher per la stampa nel video.

```
class SecretaryfindprovinceAction(Action):

    EasyCode: Explain
    def name(self) -> Text:
        return "action_find_province"

    EasyCode: Explain
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            slot_province=tracker.get_slot("province")
            print(slot_province)
            if slot_province:

                response_builder = SecretaryfindprovinceAction.Utils().get_info(slot_
                print(response_builder)
                for text in response_builder:
                    dispatcher.utter_message(text)
            else:
                dispatcher.utter_message(text="Mi serve sapere prima la Regione")

            return []
```

```
class Utils:

    EasyCode: Explain
    def build_response(self, tracker: Tracker, name:str):
        return f"Hai selezionato {name}"

    EasyCode: Explain
    def get_info(self,my_region) -> List[Dict[Text, Any]]:
        region=my_region.split("-")[0]
        province=my_region.split("-")[1]
        take_province=secretary.instance().readlistprovince(region)
        info=secretary.instance().readprovince(take_province,province)
        information_province=info.loc[0].values.tolist()
        reply=[]
        for index in information_province:
            reply.append(str(index))

        print(information_province)

        return reply
```

Figura 4.49: Action Cerca Provincia

In questa azione è stato anche inserito un controllo nel caso in cui l'utente inserisca

direttamente la provincia nell'input. Si è scelto di non bypassare la gerarchia della regione e poi della provincia, poiché la ricerca del segretario di provincia è un'opzione non necessaria. Pertanto, l'utente riceverà una risposta che richiede di inserire prima la regione e quindi ripetere l'intero processo per la ricerca del segretario della provincia.

Infine, per gestire il caso in cui un utente richiede semplicemente la ricerca del segretario senza specificare la regione, è stata creata un'altra azione che gestisce questa eccezione. In questa azione, si chiede all'utente di fornire il nome della regione del segretario che desidera cercare. Come mostrato nella Figura 4.50, una volta richiesto di inserire il nome della regione, l'input dell'utente viene preparato per l'azione di ricerca del segretario di regione.

```
class SecretaryfindAction(Action):


    def name(self) -> Text:
        EasyCode: Explain
        return "action_find_region2"

    def run(self, dispatcher: CollectingDispatcher,
            EasyCode: Explain
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            if tracker.get_slot('slot_ricerca_regione')==None:
                return [SlotSet('slot_ricerca_regione','chiesto'),FollowupAction("action_listen"), ActiveLoop("action_find_region2"
            else:
                return [FollowupAction("action_find_region"),ActiveLoop(None)]
```

Figura 4.50: Action per la gestione della ricerca del segretario di regione

4.4.3 Jaccard e Set-Containment

Prima di progettare il task per la ricerca degli allegati nel sito del SIM, è stata condotta un'analisi approfondita del contesto di utilizzo. Poiché uno degli obiettivo era quello di trovare un file attraverso i link nelle diverse sezioni del sito, e considerando che l'input dell'utente può contenere molte informazioni, il matching tra le due parti sarebbe potuto risultare complesso. Inoltre, poiché un secondo obiettivo era quello di aiutare l'utente a cercare un allegato, presumendo che egli non avesse piena conoscenza delle informazioni sul suddetto allegato, sarebbe stato probabile che le informazioni fornite risultassero incomplete. Alla luce di queste considerazioni, non si è optato per un confronto diretto tra le informazioni fornite dall'utente e quelle presenti nel database del sito. Invece, si è scelto di considerare le informazioni provenienti dall'input e dal database come due insiemi distinti e di trovare l'intersezione tra i due insiemi. A tal fine, sono stati studiati il coefficiente di Jaccard e il set-containment.

Il coefficiente di Jaccard è un indicatore che valuta la similarità tra due insiemi in base alla cardinalità degli stessi. Questo coefficiente viene calcolato come il rapporto tra la cardinalità dell'insieme di intersezione tra i due insiemi e la cardinalità della loro unione.

$$J = \frac{|Q \cap X|}{|Q \cup X|}$$

In effetti, quanto più è ampia l'intersezione tra gli insiemi, tanto maggiore sarà la cardinalità della loro intersezione e, di conseguenza, il coefficiente di Jaccard aumenterà proporzionalmente.

Tuttavia, nel calcolo del coefficiente di Jaccard, la stima della similarità tra dataset può essere distorta quando i due insiemi hanno dimensioni significativamente diverse. Nel caso in cui l'insieme Q' abbia una cardinalità notevolmente inferiore rispetto all'insieme X', come mostrato nella Figura 4.51, la cardinalità dell'intersezione risulterà molto inferiore rispetto alla cardinalità dell'unione. Di conseguenza, il coefficiente di Jaccard risulterebbe estremamente piccolo, indicando una bassa somiglianza tra i due insiemi. Inoltre, considerando il caso limite in cui X' è un superinsieme di Q', i due insiemi sarebbero empiricamente simili (poiché

X′ contiene tutti gli elementi di Q'), ma il coefficiente di Jaccard sarebbe ancora piccolo, indicando una bassa affinità. Per risolvere questo problema, si utilizza una misura più accurata chiamata set containment, che si calcola come la dimensione dell'intersezione divisa per la dimensione dell'insieme con la cardinalità maggiore.

$$Containment = \frac{|Q' \cap X'|}{max(|Q'|, |X'|)}$$

Utilizzando il set-containment, nel caso limite in cui X′ sia un superinsieme di Q′, il calcolo della misura risulterebbe avere valore massimo, indicando un'ottima affinità tra i due insiemi. Questo risolve efficacemente il limite del calcolo del coefficiente di Jaccard (Figura 4.51).

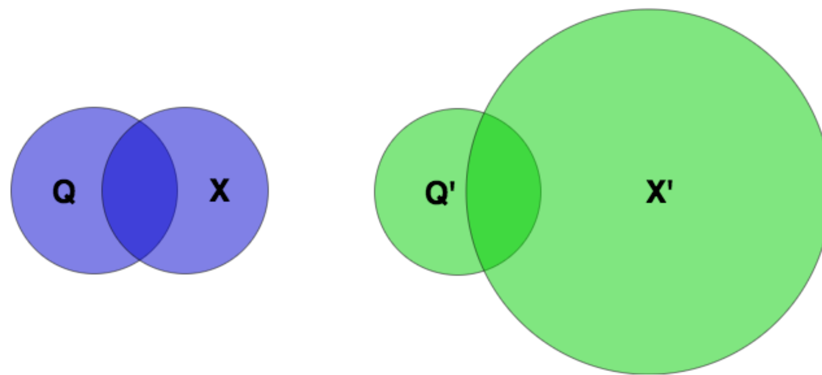

Figura 4.51: Jaccard e Set-Containment

Inoltre, considerando che Q′ rappresenta l'input dell'utente e X′ rappresenta il dominio del dataset che contiene gli allegati del sito, è evidente che il calcolo del set-containment risulti vantaggioso per gli obiettivi del progetto.

Per applicare il set-containment, è stata utilizzata la funzione *LSHEnsemble*, che sfrutta la tecnica del MinHash per confrontare la somiglianza tra i singoli valori degli insiemi per poi calcolare il set-containment.

4.4.4 Realizzazione della funzione "ricerca allegato"

Dopo aver compreso l'approccio ai dati, è stato avviato il processo di costruzione della struttura dell'azione e delle relative funzioni per soddisfare le richieste del SIM. Come già discusso nei capitoli precedenti, la prima sfida affrontata è stata la creazione di un database adattabile al progetto, poiché non si disponeva di accesso diretto al database del sito web. Dopo un'attenta analisi della struttura del sito, è stato creato manualmente un file Excel che comprende una colonna contenente tutte le aree tematiche del sito e, per ciascuna area, tutti i collegamenti agli allegati corrispondenti. L'uso di un file Excel può sembrare datato, in realtà, questa scelta è stata fatta perché, avendo una conoscenza limitata sulla struttura effettiva del database del sito, si è optato per un approccio più rapido per acquisire le informazioni necessarie dal sito stesso, attraverso le funzioni di copia e incolla.

Dopo aver costruito il database, il passo successivo è stato quello della progettazione. L'idea di base è stata quella di creare una funzione in loop eseguita fino a quando il chatbot individua un documento corrispondente oppure finchè l'utente decide di interrompere la ricerca, con il comando *fermati*. All'interno di questo loop, l'input dell'utente viene acquisito e assegnato ad uno slot, che sarà successivamente processato per l'analisi (Figura 4.52).

Il valore ottenuto dall'input viene quindi inviato alla classe *Utils*, che preelabora i dati per individuare eventuali corrispondenze.

```
class ActionSearchValidation(Action):
    EasyCode: Explain
    def name(self) -> Text:
        return "action_search_parola"

    EasyCode: Explain
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            st=tracker.get_slot("slot_ricerca_dei_documenti")
            print(st)
            #dispatcher.utter_message(f"Ecco la mia ricerca-->{st}")
            test=tracker.latest_message['text'].lower()
            #dispatcher.utter_message(text=test)
            filename="data/map/sito_map.txt"
            t=ActionSearchValidation.Utils().find_word_in_file(filename,test, dispatcher)
            print(t)
            print(f"ECcco il mio {t}")
            if not t:
                stop=tracker.latest_message['text'].lower()
                if stop=="fermati":
                    dispatcher.utter_message("Va bene mi fermo! Chiedimi altro se vuoi")
                    return[AllSlotsReset(), ActiveLoop(None)]
                else:
                    dispatcher.utter_message(text="Ok vuoi cercare un documento, allora scrivimi l'argomento che vuoi cercare")
                    dispatcher.utter_message(text="P.s. Inoltre se vuoi uscire dalla ricerca basta che mi scrivi: fermati")
                    return [FollowupAction("action_listen"),SlotSet('slot_ricerca_dei_documenti',st),ActiveLoop("action_search_paro
            else:
                ActionSearchValidation.Utils().prendi_link(t,dispatcher)
                return [AllSlotsReset(),ActiveLoop(None)]
```

Figura 4.52: Azione per la Ricerca dell'allegato

All'interno di questa classe, sono state implementate diverse funzioni per l'elaborazione e la preparazione dell'output da inviare al dispatcher. La funzione principale della classe è *find_word_in_file* (Figura 4.53).

```
def find_word_in_file(self,filename, my_input, stampante):
    # Nella riga sottostante creo il file txt per l'elenco de
    self.sito_map_txt(filename)
    not_allowed = [
                        "di",
                        "un",
                        "del",
                        "dal",
                        "dei",
                        "a",
                        "a",
                        "e",
                        "al",
                        "in",
                        "le",
                        "il",
                        "la",
                        "per"
                    ]
    # Il vettore s mi serve per poter inserire tutte le parol
    # tra l'input di testo e il file sito_map.txt
    s=[]
    with open(filename, 'r') as file:
        for line in file:
            # Elimina eventuali spazi bianchi e newline dalla
            line=re.sub(r'[-()]', ' ', line)
            line = line.strip()

           # Separa la linea in parole
            words = line.split()
            #metto tutte le parole del mio input in un array
            lt=list(re.sub(r'[-()]', ' ', my_input).split())
```

Figura 4.53: Funzione *find_word_in_file*

In questa funzione, la prima fase consiste nella creazione di un file di testo (Figura 4.57) in cui vengono inserite tutte le aree del sito web (aree tematiche), ricavata dalla prima colonna del database creato in precedenza. Successivamente, viene estratta la prima area tematica dal file appena creato, insieme all'input dell'utente, i quali vengono suddivisi in parole e sequenze di caratteri per il confronto. Il confronto tra le parole avviene tramite la funzione *SequenceMatcher*, che restituisce un valore compreso tra 0 e 1, dove 0 indica l'assenza di caratteri in comune e 1 indica che tutti i caratteri confrontati sono uguali. Per ottenere una similarità accettabile tra le due parole confrontate, il valore restituito dal *SequenceMatcher* deve essere superiore a 0.8. Questo valore è stato scelto per lasciare un piccolo margine alla possibilità che l'utente possa scrivere una parola molto simile ma non identica a quella confrontata (Figura 4.54).

```
            for item2 in words:
                item2=self.rimuovi_punto(item2)
                # con match trovo la similarità. Il valore si aggira
                # perchè voglio ridurre il margine di errore
                match = difflib.SequenceMatcher(None, item1, item2)
                similarity = match.ratio()
                if similarity>0.8:
                    if item2 not in not_allowed:
                        s.append(item2)
                        # Qui sotto elimino gli eventuali elementi c
                        # la stessa parola che di conseguenza passe
                        for elemento in s:
                            if s.count(elemento) > 1:
                                s.pop()
file.close()
temp=""
# Nel codice sottostante utilizzo Jaccard con una precisione di alme
# delle parole che ho trovato con l'insieme totale delle parole sia
t0 = time.perf_counter()
index=[]
set1 = set(s)
m1 = MinHash(num_perm=128)
for d in set1:
    m1.update(d.encode('utf8'))

with open(filename, 'r') as file:
    for number, line in enumerate(file):
        line=re.sub(r'[-()]', ' ', line)
        line=line.strip()
        words=line.split()
        set2 = set([self.rimuovi_punto(item) for item in words])

        # Qui utilizzo Jaccard ma alla fine ho deciso di utilizzare

        m2 = MinHash(num_perm=128)
        for d in set2:
            m2.update(d.encode('utf8'))
```

Figura 4.54: SequenceMatcher

Il processo di confronto delle parole dell'input dell'utente viene eseguito per tutte le parole delle aree tematiche presenti nel file. Questo passaggio serve a individuare tutte le parole rilevanti, che verranno inserite in un set di parole, per l'esecuzione successiva di *LSHEnsamble*, al fine di trovare l'allegato specifico cercato dall'utente. Prima di procedere con *LSHEnsamble*, vengono eliminate dal set tutte le parole non pertinenti alla ricerca, come, ad esempio le preposizioni. Dopo aver effettuato la pulizia, viene creato un minhash con 128 permutazioni per le parole selezionate. Il minhash permette di generare un insieme di funzioni di hash casuali, che generano un valore di hash per ciascun elemento in un insieme di dati. Successivamente, i valori di hash vengono confrontati per determinare la similarità tra due insiemi. Prima di trasformare le parole dell'area tematica in un minhash a 128 permutazioni, vengono eliminate le caratteristiche non necessarie, come punti o trattini. A questo punto, i due minhash vengono passati alla funzione LSHEnsamble. Come spiegato precedentemente, *LSHEnsamble* calcola la similarità tra due insiemi quando il valore di set-containment supera una determinata soglia. In questo caso, prima di stabilire l'esistenza o meno di similarità tra i due insiemi, questi ultimi vengono riprocessati nella funzione *LSHEnsamble* variando il valore di soglia del set-containment da 0.9 a 0.7 con un decremento di 0.019 (Figura 4.55).

```
    for number, line in enumerate(file):
        line=re.sub(r'[-()]', ' ', line)
        line=line.strip()
        words=line.split()
        set2 = set([self.rimuovi_punto(item) for item in words])

        # Qui utilizzo Jaccard ma alla fine ho deciso di utilizzare il set-containment

        m2 = MinHash(num_perm=128)
        for d in set2:
            m2.update(d.encode('utf8'))

        # Mi vado a creare tutti gli index per effettuare successivame il MinHashLshEnsamble
        index.append([number,m2,len(set2)])

file.close()
temp_threshold=0.9
found=False

if len(set1)>0:
    while temp_threshold>=0.7:
        lshensemble = MinHashLSHEnsemble(threshold=temp_threshold, num_perm=128,num_part=32)
        lshensemble.index(index)
        for key in lshensemble.query(m1, len(set1)):
            print(key)
            print(f"il valore del mio set-containment > {temp_threshold}:")
            print("Similarity TROVATA:")
            #temp=self.stampa_stringa(words)
            read_temp=open(filename, 'r', encoding='utf-8')
            line_match=read_temp.readlines()
            temp=line_match[key].strip("\n")
            print(temp)
            stampante.utter_message(text=f"Sezione:{temp}")
            found=True
        if found:
            break
        temp_threshold=temp_threshold-0.019
```

Figura 4.55: *LSHEnsamble*

Se non viene trovata alcuna similarità all'interno di questo intervallo di valori, si procede con la successiva area tematica e si ripetono gli stessi passaggi. Questa procedura viene ripetuta per tutte le aree tematiche. Nel caso in cui sia identificata una similarità, viene selezionata l'area tematica corrispondente e, successivamente, vengono prese tutte le informazioni relative ai link degli allegati di quell'area tematica. Se, non viene ottenuta alcuna similarità tra i due insiemi, prima di informare l'utente che nessun allegato è stato trovato, viene effettuato un ragionamento simile a quello descritto in precedenza, ma senza l'utilizzo di minhash e *LSHEnsamble*. Al loro posto viene calcolato il coefficiente di Jaccard per confrontare i due insiemi. Questo tipo di confronto (Figura 4.56) viene utilizzato poiché potrebbe verificarsi una situazione in cui l'insieme delle parole selezionate sia più grande dell'insieme delle parole dell'area tematica analizzata, rendendo *LSHEnsamble* inefficace. Per il coefficiente di Jaccard, è stata scelta una soglia di 0.5, il che significa che, se almeno la metà delle parole nell'insieme di quelle selezionate è presente nell'insieme delle parole dell'area tematica, la similarità è considerata accettabile e si può concludere che l'utente stava cercando gli allegati di quell'area tematica.

```
    if not found:
        with open(filename, 'r') as file:
            for number, line in enumerate(file):
                line=re.sub(r'[-()]', ' ', line)
                line=line.strip()
                words=line.split()
                set2 = set([self.rimuovi_punto(item) for item in words])
                jaccard_similarity = len(set1.intersection(set2)) / len(set1.union(set2))

                if jaccard_similarity>=0.5:
                    print("Similarity TROVATA Jaccard:", jaccard_similarity)
                    read_temp=open(filename, 'r', encoding='utf-8')
                    line_match=read_temp.readlines()
                    temp=line_match[number].strip("\n")
                    print(temp)
                    stampante.utter_message(text=f"Sezione:{temp}")


t1 = time.perf_counter()
print("La mia funzione ci ha messo", t1-t0, "secondi")
print(f"Queste sono le parole che ho torovato in comune nell'archivio {s}")
#return s
return temp
```

Figura 4.56: Funzione per il calcolo del coefficente di Jaccard

Al termine di tutto il processo, se viene identificata una similarità tra le diverse valutazioni effettuate, il chatbot comunicherà all'utente, mediante una serie di messaggi distinti, l'area tematica individuata e i relativi link agli allegati correlati all'area tematica. Nel caso in cui sia individuata un'area tematica ma non siano disponibili allegati, il chatbot informerà l'utente con il nome dell'area tematica individuata, aggiundendo che in essa non sono presenti allegati (Figura 4.57).

```
def prendi_link(self,corrispondenza,stampante):
    # In questo caso creo un dataframe dove le mie ricerche sono gli indici delle righe attraver
    df = pd.read_excel('data/map/Sito_map.xlsx', index_col="Ricerca")
    # Qui attraverso la chiave corrispondenza cioè cosa ho trovato mi prendo tutti i link che so
    # Per il futuro puoi utilizzare il try e chatch per gestire eventuali errori
    lst_link=df.loc[corrispondenza]
    check_empty_link=len(lst_link)
    # flag_stampa serve solo per stampare una sola volta la frase "Qui sotto..."
    flag_stampa=0
    for i in lst_link:
        if pd.isna(i):
            check_empty_link -= 1
            if check_empty_link==0:
                stampante.utter_message("Mi dispiace ma non è presente nessun allegato")
        else:
            if flag_stampa==0:
                stampante.utter_message(text="Qui sotto ti elenco gli allegati che ho trovato")
                stampante.utter_message(i)
                flag_stampa=1
            else:
                stampante.utter_message(i)

EasyCode: Explain
def sito_map_txt(self,filename):
    # Semplice funzione che mi crea il file sito_map.txt
    df=pd.read_excel('data/map/Sito_map.xlsx')
    lista_percorsi=df['Ricerca'].tolist()
    with open(filename, "w", encoding='UTF-8') as file:
        for i in lista_percorsi:
            file.write(str(i) + "\n")
    file.close()
```

Figura 4.57: Funzione per ottenere i link degli allegati e il file txt

CAPITOLO 5

Implementazione

Nel capitolo corrente, dopo aver descritto le metodologie e le fasi di progettazione, verranno trattati e illustrati tutti i passaggi per il funzionamento del chatbot. Si inizierà mostrando la sua prima attivazione e i relativi servizi associati, fornendo una chiara spiegazione dei comandi corrispondenti per le diverse funzionalità. Successivamente, verrà descritta l'implementazione del chatbot nel contesto del web, essenziale per le richieste provenienti dal SIM, poiché il chatbot deve essere in grado di operare sul loro sito internet. Inoltre, si introdurrà e verrà spiegata la possibilità di implementare il chatbot nelle app di messaggistica più comuni, fornendo una guida dettagliata dei passaggi necessari per l'app di messaggistica Telegram.

5.1 Inizializzazione del chatbot

placeins Completata la fase di progettazione e dopo aver strutturato il chatbot per soddisfare le specifiche richieste del SIM, è necessario presentare le dinamiche di funzionamento e, di conseguenza, avviare il chatbot per i test iniziali. In riferimento al'capitolo precedente (paragrafo (4.1.2)), una volta giunti al punto 8 della procedura di installazione corretta, si avrà di fronte una schermata contentente (Figura 5.1): a sinistra, la variabile *env* attiva (Punto 1 della Figura 5.1), mentre a destra il percorso in cui sono presenti i file del chatbot (Punto 2 della Figura 5.1).

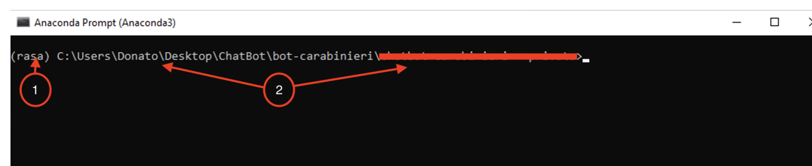

Figura 5.1: Posizione del percorso del chatbot

Ripartendo da ques'ultimo punto, il primo passo da svolgere consiste nella creazione di un modello che consenta al chatbot di comprendere gli intenti dell'utente e di instaurare una conversazione. Per ottenere ciò, è necessario eseguire l'addestramento mediante il comando *rasa train*. Prima di avviare il processo di addestramento del modello, è importante selezionare il numero di epoche da utilizzare.

Durante la fase di addestramento di un modello di machine learning, le epoche rappresentano il numero di volte in cui l'intero set di dati di addestramento viene presentato al modello. In altre parole, un'epoca corrisponde a un ciclo completo dove vengono passati tutti i dati di addestramento. Durante ogni epoca, il modello esegue una serie di iterazioni, in

cui predice gli output in base ai dati di input e aggiorna i suoi parametri interni in base agli errori commessi.

L'addestramento di un modello di solito prevede più epoche per consentire al modello stesso di apprendere dai dati in modo progressivo e migliorare le sue prestazioni. Ad ogni epoca, il modello regola i suoi pesi e i suoi parametri interni per ridurre l'errore tra le sue previsioni e i valori desiderati. Il numero di epoche da utilizzare dipende dal problema specifico, dalle dimensioni del dataset e dalla complessità del modello. Un numero troppo basso di epoche potrebbe non consentire al modello di convergere verso una soluzione ottimale, mentre un numero troppo elevato potrebbe causare l'overfitting, ovvero una memorizzazione eccessiva dei dati di addestramento senza una buona capacità di generalizzazione.

Nel contesto del progetto, è stata adottata una configurazione di 200 epoche per il processo di addestramento. Per impostare questo valore nel progetto, è necessario accedere al file *config.yml* (Figura 5.2).

Figura 5.2: Numero di epoche-*config.yml*

Il comando *rasa train* avvia il processo di addestramento dei modelli NLU e Core utilizzati dal chatbot. Rasa Core si occupa della gestione del dialogo, cioè di capire l'intenzione dell'utente e di decidere quale azione intraprendere in risposta. Durante l'addestramento, il sistema viene esposto a un set di dati contenenti esempi di input di un utente e relative etichette o azioni corrispondenti. Il modello di Rasa apprende da questi dati al fine di comprendere e rispondere adeguatamente alle richieste degli utenti.

Nel processo di l'addestramento, i parametri del modello vengono ottimizzati per massimizzare la sua capacità di classificazione e generazione di risposte coerenti. Una volta completato l'addestramento, il modello può essere utilizzato per interagire in modo efficace con gli utenti.

Una volta completato il processo di addestramento, verrà generato un file compresso (*.tar.gz*) che sarà posizionato nella cartella *models* del progetto (Figura 5.3).

```
Your Rasa model is trained and saved at 'models\20230625-213058-colorful-journal.tar.gz'.
```

Figura 5.3: Crezione di un modello in Rasa

Una volta creato il modello, è possibile procedere con l'inizializzazione del chatbot utilizzando il comando *rasa shell*.

Inviato il comando, verrà avviata una sessione interattiva in cui si possono inviare messaggi di testo simulando l'interazione con il chatbot.

Durante la sessione con la shell, si possono inviare messaggi al chatbot e ricevere risposte simulate in tempo reale. E' possibile testare e valutare il comportamento del chatbot, verificare

come gestisce le domande e fornire risposte adeguate. La shell consente di esplorare le capacità del proprio chatbot, verificare la corretta configurazione delle risposte e rilevare eventuali problemi o errori.

Con l'utilizzo della shell, inoltre, si possono eseguire le azioni personalizzate definite nel proprio progetto. Per poterle testare o eseguire è necessario avviare un servizio dedicato. A tale scopo, bisognerà:

1. Aprire un altro terminale in Anaconda, riattivare la variabile *env* e navigare verso la cartella contenente i file del chatbot (Figura 5.4).

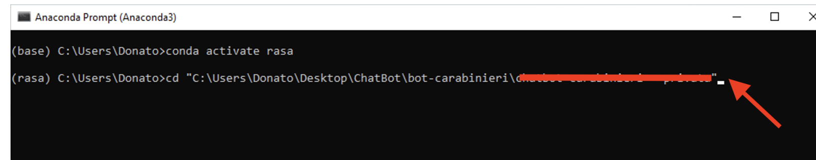

Figura 5.4: Nuova shell per l'avvio del server delle azioni personalizzate

2. Eseguire il comando *rasa run actions* (Figura 5.5).

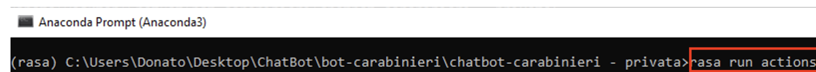

Figura 5.5: Comando *rasa run actions*

Dopo aver premuto invio, tutte le azioni saranno caricate nel server locale (Figura 5.6).

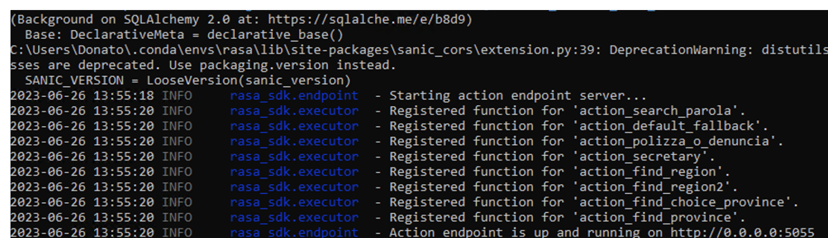

Figura 5.6: Comando *Server per le azioni personalizzate*

3. Per attivare il server, è necessario verificare se l'istruzione *action_endpoint* nel file *endpoint.yml* (come mostrato nella figura 5.7) è decommentata. Nel caso in cui sia commentata, sarà necessario rimuovere il commento.

```
action_endpoint:
  url: "http://localhost:5055/webhook"
```

Figura 5.7: Istruzione per il funzionamento del server delle action

L'*action_endpoint* in Rasa è un'opzione di configurazione che definisce l'endpoint (URL) a cui inviare le richieste delle azioni personalizzate del chatbot. Inoltre, consente di separare la logica delle azioni personalizzate dal resto del sistema e permette di sfruttare servizi esterni per l'esecuzione di azioni specifiche.

Durante la fase di testing e controllo della qualità della logica del chatbot sviluppato, è possibile sfruttare altri comandi oltre a *rasa shell* nella shell iniziale.

Comandi come *rasa shell –debug* e *rasa interactive* consentono di monitorare le diverse fasi di una conversazione con l'utente, allo scopo di verificare il corretto funzionamento della logica del chatbot.

In particolare, *rasa shell –debug* (Figura 5.8) avvia la modalità di debug del chatbot. Questa modalità offre ulteriori informazioni e registra le interazioni dell'utente con il chatbot.

All'avvio del chatbot, durante l'interazione con l'utente, vengono visualizzati i messaggi di input dell'utente (Punto 1 della Figura 5.8), i messaggi di output del chatbot (Punto 3 della Figura 5.8), le previsioni degli intenti e le azioni eseguite (Punto 2 della Figura 5.8). Inoltre, vengono mostrati anche i punteggi di confidenza delle previsioni degli intenti e delle azioni.

La modalità di debug è particolarmente utile durante lo sviluppo e il test del chatbot in quanto fornisce una panoramica dettagliata di come il chatbot sta elaborando le richieste degli utenti e prendendo decisioni. Ciò consente di identificare eventuali problemi, comprendere il flusso di conversazione e verificare il corretto funzionamento delle regole e delle azioni personalizzate.

```
Bot loaded. Type a message and press enter (use '/stop' to exit):
Your input ->  ciao che fai?
2023-06-26 16:01:04 DEBUG    rasa.core.lock_store  - Issuing ticket for conversation 'a365914ad8a94608901477e1c59f79b1'.
2023-06-26 16:01:04 DEBUG    rasa.core.lock_store  - Acquiring lock for conversation 'a365914ad8a94608901477e1c59f79b1'.
2023-06-26 16:01:04 DEBUG    rasa.core.lock_store  - Acquired lock for conversation 'a365914ad8a94608901477e1c59f79b1'.
2023-06-26 16:01:04 DEBUG    rasa.core.tracker_store  - Could not find tracker for conversation ID 'a365914ad8a946089014
77e1c59f79b1'.
2023-06-26 16:01:04 DEBUG    rasa.core.processor  - Starting a new session for conversation ID 'a365914ad8a94608901477e1
c59f79b1'.
2023-06-26 16:01:04 DEBUG    rasa.core.processor  - Policy prediction ended with events '[]'.
2023-06-26 16:01:04 DEBUG    rasa.core.processor  - Action 'action_session_start' ended with events '[<rasa.shared.core.
events.SessionStarted object at 0x000001E812967460>, ActionExecuted(action: action_listen, policy: None, confidence: Non
e)]'.
2023-06-26 16:01:04 DEBUG    rasa.core.processor  - Current slot values:
        documenti: None
        slot_ricerca_dei_documenti: None
        slot_ricerca_regione: None
        province: None
        choice_province: None
        regione: None
        session_started_metadata: None
2023-06-26 16:01:04 DEBUG    rasa.engine.runner.dask  - Running graph with inputs: {'__message__': [<rasa.core.channels.
channel.UserMessage object at 0x000001E812937D30>]}, targets: ['run_RegexMessageHandler'] and ExecutionContext(model_id=
'58f2a1c61317440381bfb0299e5d7174', should_add_diagnostic_data=False, is_finetuning=False, node_name=None).
2023-06-26 16:01:04 DEBUG    rasa.engine.graph  - Node 'nlu_message_converter' running 'NLUMessageConverter.convert_user
_message'.
```

```
2023-06-26 16:01:06 DEBUG    rasa.core.policies.ensemble  - Predicted next action using RulePolicy.
2023-06-26 16:01:06 DEBUG    rasa.core.processor  - Predicted next action 'action_secretary' with confidence 1.00.
2023-06-26 16:01:06 DEBUG    rasa.core.actions.action  - Calling action endpoint to run action 'action_secretary'.
2023-06-26 16:01:08 DEBUG    rasa.core.processor  - Policy prediction ended with events '[<rasa.shared.core.events.Defin
ePrevUserUtteredFeaturization object at 0x000001E8481A5B50>]'.
2023-06-26 16:01:08 DEBUG    rasa.core.processor  - Action 'action_secretary' ended with events '[BotUttered('Ciao sono
il tuo BotComander,e sono qui per servirti nel miglior modo possibile.
', {"elements": null, "quick_replies": null, "buttons": null, "attachment": null, "image": null, "custom": null}, {}, 16
87788068.3572776), BotUttered('Per ora sono solo addestrato per poterti guidara a:
', {"elements": null, "quick_replies": null, "buttons": null, "attachment": null, "image": null, "custom": null}, {}, 16
87788068.3572776), BotUttered('-Recuperare la password
', {"elements": null, "quick_replies": null, "buttons": null, "attachment": null, "image": null, "custom": null}, {}, 16
87788068.3572776), BotUttered('-Iscriverti al sito
', {"elements": null, "quick_replies": null, "buttons": null, "attachment": null, "image": null, "custom": null}, {}, 16
87788068.3572776), BotUttered('-Cercare il Segretario di Regione', {"elements": null, "quick_replies": null, "buttons":
null, "attachment": null, "image": null, "custom": null}, {}, 1687788068.3572776), BotUttered('-Procedura per il Trattam
ento Stipendiale e Pensionistico', {"elements": null, "quick_replies": null, "buttons": null, "attachment": null, "image
": null, "custom": null}, {}, 1687788068.3572776), BotUttered('-Cercare un allegato', {"elements": null, "quick_replies"
: null, "buttons": null, "attachment": null, "image": null, "custom": null}, {}, 1687788068.3572776), BotUttered('-Gesti
one della polizza e dei sinistri', {"elements": null, "quick_replies": null, "buttons": null, "attachment": null, "image
": null, "custom": null}, {}, 1687788068.3572776)]'.
2023-06-26 16:01:08 DEBUG    rasa.core.processor  - Current slot values:
        documenti: None
        slot_ricerca_dei_documenti: None
        slot_ricerca_regione: None
        province: None
        choice_province: None
        regione: None
        session_started_metadata: None
2023-06-26 16:01:08 DEBUG    rasa.engine.runner.dask  - Running graph with inputs: {'__tracker__': <rasa.shared.core.tra
```

```
s'.
2023-06-26 16:01:08 DEBUG    rasa.core.policies.rule_policy  - Current tracker state:
[state 1] user intent: saluto | previous action name: action_listen
[state 2] user intent: saluto | previous action name: action_secretary
2023-06-26 16:01:08 DEBUG    rasa.core.policies.rule_policy  - There is a rule for the next action 'action_listen'.
2023-06-26 16:01:08 DEBUG    rasa.engine.graph  - Node 'run_TEDPolicy3' running 'TEDPolicy.predict_action_probabilities'
.
2023-06-26 16:01:08 DEBUG    rasa.core.policies.ted_policy  - TED predicted 'action_listen' based on user intent.
2023-06-26 16:01:08 DEBUG    rasa.engine.graph  - Node 'run_UnexpecTEDIntentPolicy2' running 'UnexpecTEDIntentPolicy.pre
dict_action_probabilities'.
2023-06-26 16:01:08 DEBUG    rasa.core.policies.unexpected_intent_policy  - Skipping predictions for UnexpecTEDIntentPol
icy as either there is no event of type `UserUttered`, event's intent is new and not in domain or there is an event of t
ype `ActionExecuted` after the last `UserUttered`.
2023-06-26 16:01:08 DEBUG    rasa.engine.graph  - Node 'select_prediction' running 'DefaultPolicyPredictionEnsemble.comb
ine_predictions_from_kwargs'.
2023-06-26 16:01:08 DEBUG    rasa.core.policies.ensemble  - Predicted next action using RulePolicy.
2023-06-26 16:01:08 DEBUG    rasa.core.processor  - Predicted next action 'action_listen' with confidence 1.00.
2023-06-26 16:01:08 DEBUG    rasa.core.processor  - Policy prediction ended with events '[]'.
2023-06-26 16:01:08 DEBUG    rasa.core.processor  - Action 'action_listen' ended with events '[]'.
2023-06-26 16:01:08 DEBUG    rasa.core.lock_store  - Deleted lock for conversation 'a365914ad8a94608901477e1c59f79b1'.
Ciao sono il tuo BotComander,e sono qui per servirti nel miglior modo possibile.
Per ora sono solo addestrato per poterti guidara a:
-Recuperare la password
-Iscriverti al sito
-Cercare il Segretario di Regione
-Procedura per il Trattamento Stipendiale e Pensionistico
-Cercare un allegato
-Gestione della polizza e dei sinistri
Your input ->
```

Figura 5.8: Funzionamento di *rasa_debug*

Il comando *rasa interactive* (Figure 5.9 e 5.10) avvia la modalità interattiva di addestramento del chatbot. Tale modalità consente di interagire in tempo reale con il chatbot e permette di fornire feedback sulle sue risposte, consentendo di migliorare il suo comportamento nel corso del tempo.

L'utente attraverso i feedback sulle risposte del chatbot (Punto 2 della Figura 5.10), può correggere o suggerire alternative più appropriate (Punto 3 della Figura 5.10). Inoltre, durante la modalità interattiva, vengono mostrate le previsioni degli intenti, le risposte generate dal chatbot e l'elenco delle azioni possibili (Figure 5.9 e 5.10). Il feedback dato dall'utente viene registrato e utilizzato per addestrare il modello del chatbot in modo incrementale.

L'uso di tale modalità è particolarmente utile per migliorare rapidamente le risposte del chatbot in base al feedback diretto degli utenti. Terminata la modalità, verrà generato il file del nuovo modello.

Infine, è possibile visualizzare il diagramma di flusso (Fiugra 5.11) della conversazione aprendo un qualsiasi browser e collegandosi al link (Figura 5.12) che verrà fornito nel terminale dopo aver eseguito il comando *rasa interactive*.

Figura 5.9: Funzionamento di *rasa_interactive*-1

```
possibile.
Per ora sono solo addestrato per poterti guidara a:
-Recuperare la password
-Iscriverti al sito
-Cercare il Segretario di Regione
-Procedura per il Trattamento Stipendiale e Pensionistico
-Cercare un allegato
-Gestione della polizza e dei sinistri
action_listen 0.00
6                                                        Ciao, cosa sapresto fare?
                                                              intent: saluto 0.97
7   action_secretary 1.00
    Ciao sono il tuo BotComander,e sono qui per servirti nel miglior modo
    possibile.
    Per ora sono solo addestrato per poterti guidara a:
    -Recuperare la password
    -Iscriverti al sito
    -Cercare il Segretario di Regione                          1
    -Procedura per il Trattamento Stipendiale e Pensionistico
    -Cercare un allegato
    -Gestione della polizza e dei sinistri                                 Current slots
:
    documenti: None, slot_ricerca_dei_documenti: None, slot_ricerca_regione: None, province: None, choice_province:
None, regione: None, session_started_metadata: None

------
```

```
Per ora sono solo addestrato per poterti guidara a:
-Recuperare la password
-Iscriverti al sito
-Cercare il Segretario di Regione
-Procedura per il Trattamento Stipendiale e Pensionistico
-Cercare un allegato
-Gestione della polizza e dei sinistri
action_listen 0.00
8                                                        Ciao, cosa sapresti fare?
                                                              intent: saluto 0.97
9   action_secretary 1.00
    Ciao sono il tuo BotComander,e sono qui per servirti nel miglior modo
    possibile.
    Per ora sono solo addestrato per poterti guidara a:
    -Recuperare la password
    -Iscriverti al sito
    -Cercare il Segretario di Regione
    -Procedura per il Trattamento Stipendiale e Pensionistico
    -Cercare un allegato
    -Gestione della polizza e dei sinistri                                 Current slots
:
    documenti: None, slot_ricerca_dei_documenti: None, slot_ricerca_regione: None, province: None, choice_province:
None, regione: None, session_started_metadata: None

------
? The bot wants to run 'action_listen', correct? No    2
```

```
? What is the next action of the bot? (Use arrow keys)
» <create new action>
  1.00 action_listen                 3
  0.70 action_default_fallback
  0.00 ...
  0.00 action_back
  0.00 action_deactivate_loop
  0.00 action_default_ask_affirmation
  0.00 action_default_ask_rephrase
  0.00 action_extract_slots
  0.00 action_find_choice_province
  0.00 action_find_province
  0.00 action_find_region
  0.00 action_find_region2
  0.00 action_polizza_o_denuncia
  0.00 action_restart
  0.00 action_revert_fallback_events
  0.00 action_search_parola
  0.00 action_secretary
  0.00 action_session_start
  0.00 action_two_stage_fallback
  0.00 action_unlikely_intent
  0.00 utter_arrivederci
  0.00 utter_iamnotabot
  0.00 utter_iscrizione
  0.00 utter_previdenziale
  0.00 utter_reset_password
  0.00 utter_saluto
  0.00 utter_scelta_trattamento_pensioninstico
  0.00 utter_segretario
```

Figura 5.10: Funzionamento di *rasa_interactive*-2

Figura 5.11: Esempio di un diagramma di flusso di una conversazione

Figura 5.12: Link per il diagramma di flusso

5.2 Implementazione web del chatbot

Dopo tutto il lavoro svolto fino a questo punto, l'obiettivo finale è l'utilizzo del chatbot all'interno del sito web del Sindacato Italiano dei Carabinieri. A tal fine, è stato studiato e creato un metodo per consentirne l'implementazione nel portale web. Per quanto riguarda la progettazione della parte web, si è deciso di utilizzare un frontend di un'applicazione web per chatbot già predisposto per l'integrazione con i servizi di Rasa.

Il frontend è la parte di un'applicazione o di un sistema informatico che gestisce l'interazione e la presentazione dell'informazione agli utenti. È responsabile per la creazione di un'interfaccia utente intuitiva e interattiva attraverso la quale gli utenti possono interagire con l'applicazione o il sistema. L'integrazione del chatbot con il lato web è possibile grazie al fatto che Rasa, e i suoi servizi correlati, operano utilizzando richieste di tipo GET e POST.

Le richieste di GET e POST sono due metodi fondamentali nel protocollo HTTP (Hypertext Transfer Protocol) per la comunicazione tra client e server. La richiesta GET viene utilizzata per ottenere risorse dal server inviando una richiesta attraverso un URL (una stringa di testo che specifica la posizione di una risorsa su Internet e consente ai browser e ad altre applicazioni di accedere a tale risorsa). Questo metodo è ampiamente utilizzato quando si tratta di recuperare informazioni, come pagine web o dati specifici, dal server.

D'altra parte, la richiesta POST viene utilizzata per inviare dati al server, come informazioni di un modulo compilato o una richiesta di elaborazione di una risorsa. I dati vengono inclusi nel corpo della richiesta POST, rendendola ideale per l'invio di informazioni sensibili sul server.

In Rasa, queste richieste GET e POST vengono utilizzate per diverse funzionalità. Ad esempio, durante l'addestramento del modello di linguaggio, è possibile utilizzare una richiesta POST per inviare i dati di addestramento al server di Rasa. Inoltre, Rasa utilizza richieste GET e POST anche per l'interazione con l'API del modello di linguaggio durante la fase di chat. Le richieste GET vengono utilizzate per ottenere informazioni sullo stato del modello, mentre le richieste POST vengono utilizzate per inviare messaggi degli utenti al modello e ricevere le risposte corrispondenti.

In entrambi i casi, le richieste GET e POST svolgono un ruolo fondamentale nell'interazione tra Rasa e il server, consentendo la comunicazione dei dati e l'elaborazione delle richieste per fornire risposte adeguate agli utenti.

Il frontend del chatbot è costituito da due file distinti:

1. *index.html*
2. *index.js*

Prima di esaminare il primo file, si fornirà una breve introduzione del secondo file. Il file *index.js* costituisce il nucleo del frontend del chatbot e contiene diverse funzioni, tra cui le animazioni del chatbot, il design e le funzioni di connessione al server. Questo file è stato sviluppato utilizzando il linguaggio di programmazione JavaScript.

JavaScript è un linguaggio di programmazione ad alto livello, interpretato e orientato agli oggetti, utilizzato principalmente per aggiungere interattività e funzionalità dinamiche alle pagine web. Poiché *index.js* contiene un gran numero di funzioni che non sono state studiate nel dettaglio, non ci soffermeremo molto su di esso, lasciando ulteriori approfondimenti per progetti futuri. Passando al primo file, ossia *index.html*, invece, spiegheremo dettagliatamente la sua creazione. Questo file è stato creato per fornire un esempio di integrazione del chatbot in qualsiasi sito web senza richiedere operazioni invasive o adattamenti specifici al sito stesso. Come si evince dall'estensione del file, esso è stato realizzato utilizzando HTML. HTML (HyperText Markup Language) è il linguaggio di base per la creazione di siti web e viene interpretato dai browser per visualizzare i contenuti sullo schermo.

Ogni pagina HTML è costituita da un insieme di elementi, organizzati in una struttura ad albero gerarchica. Inoltre, HTML utilizza una serie di tag per definire la struttura e il significato dei diversi elementi all'interno di una pagina web. I tag sono circondati da parentesi angolari ("<>" e "</>") e forniscono informazioni sul tipo di elemento, come testo, immagini, link, tabelle, formulari e altro.

In particolare, come mostrato nella Figura 5.13, viene costruita una semplice pagina web utilizzando le istruzioni *<!doctype html>*, *<html>*, *<head>*, *<title>*, *<body>* e i rispettivi tag di chiusura. Successivamente, all'interno del tag *<body>*, viene aperto un tag *<script>* e vengono inserite alcune righe di codice in JavaScript (evidenziate con un rettangolo rosso nella Figura 5.13). Le righe di codice sono fondamentali per visualizzare la schermata del chatbot sul dispositivo. In particolare, forniamo una descrizione dettagliata delle righe seguenti:

- Nella riga 9, tramite l'istruzione *src* presente nella riga 8, si richiama il file *index.js*. Questa operazione è molto importante poiché il file *index.js* contiene il codice necessario per il funzionamento del chatbot nel contesto web.
- Nella riga 12, con l'istruzione *window.WebChat.default*, vengono richiamate tutte le funzioni presenti nel file *index.js* al fine di eseguire il chatbot sul lato web, passando ad esso alcuni parametri come:
  - l'intento iniziale da caricare (riga 14), che specifica l'intento predefinito che verrà caricato quando il chatbot viene visualizzato sulla pagina web;
  - la lingua del chatbot (riga 15), che determina la lingua utilizzata nelle risposte e nell'interazione con gli utenti;
  - il link del server da cui (riga 16), che indica l'indirizzo del server a cui il chatbot invierà le richieste e riceverà le risposte;
  - il titolo del chatbot (riga 17), che fornisce un titolo descrittivo per il chatbot sulla pagina web;
  - il sottotitolo del chatbot (riga 18), che fornisce ulteriori informazioni o istruzioni per l'utente.
- L'istruzione *localStorage.clear()* (riga 26) è una funzione che viene utilizzata per pulire la cronologia della conversazione del chatbot quando la pagina web in cui è presente viene chiusa o ricaricata. Questo assicura che la cronologia precedente non venga mantenuta quando si avvia una nuova sessione del chatbot.

```
<!doctype html>
<html lang="it">
<head><title>Ciao Chatbot!</title></head>
<body>
  Ask EasyCode | Explain | Refactor
  <script>!(function () {
    let e = document.createElement("script"),
      t = document.head || document.getElementsByTagName("head")[0];
    (e.src =
      "index.js"),
      (e.async = !0),
      Ask EasyCode | Explain | Refactor
      (e.onload = () => {
        window.WebChat.default(
          {
            initPayload : '/saluto',
            customData: { language: "it" },
            socketUrl: "http://localhost:5005",
            title: 'Chat Assistenza SimCarabinieri',
            subtitle: 'Assistenza sito internet'
          },
          null
        );
      }),
      t.insertBefore(e, t.firstChild);
  })();
  // Pulire la cache della chat
  localStorage.clear()
  </script>
</body>
</html>
```

Figura 5.13: File *index.html*

5.3 Implementazione del chatbot su Telegram

In questa sezione, vengono illustrate le procedure necessarie per implementare il chatbot all'interno dell'app di messaggistica Telegram. Anche se questa procedura non fa parte delle specifiche assegnate, è stata eseguita per valorizzare ulteriormente il lavoro svolto, cercando di evidenziare il concetto di portabilità del chatbot, in quanto l'app di messaggistica può essere installata su smartphone. Il primo passo consiste nella creazione di una sezione dedicata al chatbot all'interno dell'applicazione di Telegram. Le fasi per l'inizializzazione dell'area che ospiterà il chatbot sono le seguenti:

1. Aprire l'applicazione di Telegram;
2. Scrivere nella barra di ricerca; *botfather* (Indicato nella Figura 5.14)
3. Avviare la conversazione scrivendo come messaggio di testo */start* (Punto 1 della Figura 5.15). A questo punto il bot rispondera illustrando tutte le sue funzionalità (Punto 2 della Figura 5.15);
4. Scrivere in risposta nell'apposito campo di input */newbot*. Premuto invio, il bot chiederà di scrivere il nome del chatbot (Punto 1 e 2 della Figura 5.16);
5. Scrivere il nome desiderato del proprio chatbot e premere invio. A questo punto, il bot fornirà il token di accesso per le HTTP API. Tale token permette di comunicare con i server di telegram (Punto 3 e 4 della Figura 5.16).

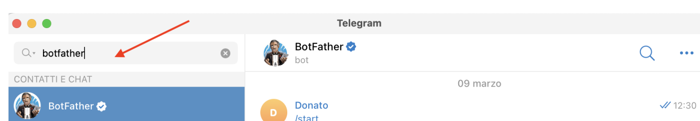

Figura 5.14: Procedura di registrazione del chatbot su Telegram-1

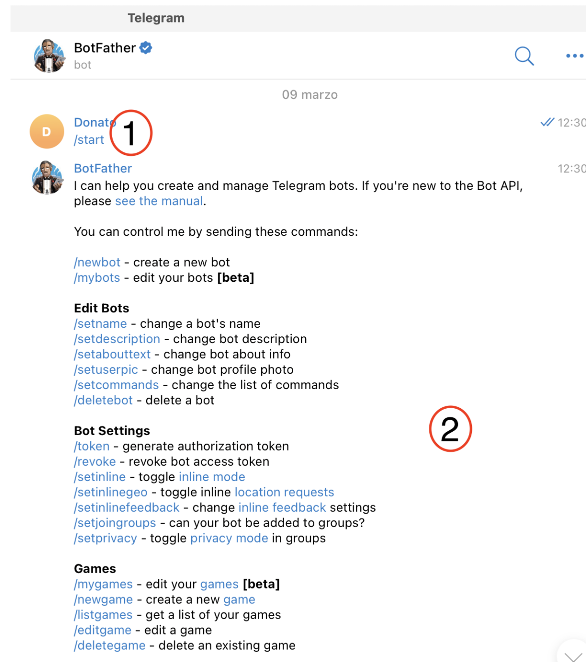

Figura 5.15: Procedura di registrazione del chatbot su Telegram-2

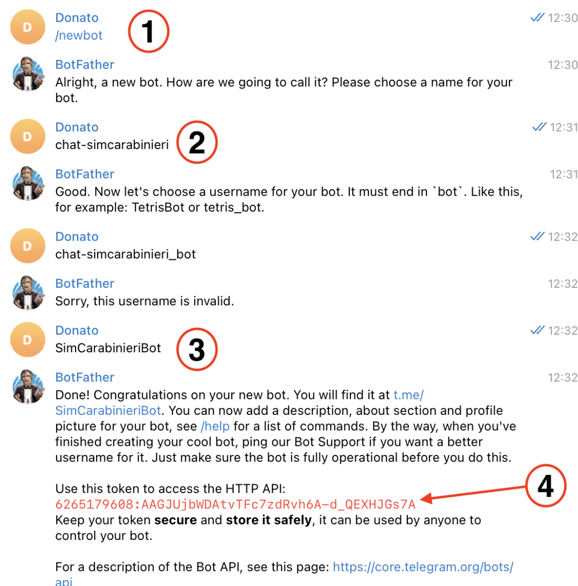

Figura 5.16: Procedura di registrazione del chatbot su Telegram-3

Una volta ottenuto il token di accesso, il passo successivo consiste nell'abilitare il servizio webhooks nel file *credentials.yml* del progetto. In Rasa, i webhook sono dei meccanismi che consentono al chatbot di comunicare con sistemi esterni, come applicazioni web o servizi di messaggistica.

Un webhook riceve richieste dai messaggi inviati dagli utenti al chatbot e può essere utilizzato per eseguire azioni personalizzate o integrare il chatbot con altre applicazioni. Ad esempio, un webhook può essere utilizzato per inviare dati a un database, eseguire operazioni di backend complesse o integrarsi con servizi esterni per recuperare informazioni aggiuntive da mostrare agli utenti.

In sostanza, i webhook consentono al chatbot di andare oltre le funzionalità predefinite,

permettendo un'interazione più dinamica e personalizzata con l'ambiente esterno. Per attivare il servizio, è sufficiente inserire poche istruzioni nella sezione *rest* del file *credentials.yml* (Fiugra 5.17).

```
rest:
#  # you don't need to provide anything here - this channel doesn't
#  # require any credentials

telegram:
  verify: "SimCarabinieriBot"
  access_token: "6265179608:AAGJUjbWDAtvTFc7zdRvh6A-d_QEXHJGs7A"
  webhook_url: "https://8e0a-178-22-250-237.eu.ngrok.io/webhooks/telegram/webhook"


socketio:
  user_message_evt: user_uttered
  bot_message_evt: bot_uttered
  session_persistence: true
```

Figura 5.17: Attivazione del servizio per l'app di messaggistica Telegram

Le righe di codice della Figura 5.17, relative alla sezione di Telegram, specificano le seguenti istruzioni:

- *verify*: nome del chatbot;
- *access_token*: token di accesso fornito da Telegram;
- *webhook_url*: indirizzo per il collegamento al server in cui sono ospitati i servizi webhook.

Invece, le righe di codice della Figura 5.17, relative alla sezione di *socketio*, indicano:

- *user_message_evt: user_message_evt:user_uttered*: gestisce gli eventi associati ai messaggi utente nel contesto di un'applicazione specifica;
- *bot_message_evt: bot_uttered*: gestisce gli eventi associati ai messaggi del chatbot nel contesto di un'applicazione specifica;
- *session_persistence: true*: viene utilizzata per abilitare la persistenza della sessione. Quest'ultima propietà è particolarmente utile in scenari in cui le conversazioni sono lunghe e complesse, in modo che il chatbot possa mantenere la memoria di ciò che è stato discusso in precedenza.

Infine, per poter utilizzare il chatbot su un sito web o sull'app di Telegram, è necessario eseguire un ultimo passo fondamentale, ovvero mantenere aperte due shell di Anaconda. Dopo aver attivato la variabile *env* e dopo essersi posizionati nella cartella del chatbot in entrambe le shell, si dovrà lanciare il comando *rasa run actions* in una delle shell, mentre nell'altra si eseguirà il comando *rasa run –enable-api –cors "*"*. Quest'ultimo viene utilizzato per avviare un server API per il chatbot creato con Rasa. Tale comando consente al chatbot di essere accessibile tramite API, consentendo ad altre applicazioni o servizi di interagire con esso.

Dopo aver eseguito il comando, per verificare il corretto funzionamento, è necessario controllare che nella shell venga visualizzata la frase: *Rasa server is up and running*.

```
  SANIC_VERSION = LooseVersion(sanic_version)
2023-06-28 11:27:08 INFO     root  - Starting Rasa server on http://0.0.0.0:5005
2023-06-28 11:27:10 INFO     rasa.core.processor  - Loading model models\20230628-112405-wintry-heron.tar.gz...
2023-06-28 11:28:12 WARNING  rasa.shared.utils.common  - The UnexpecTED Intent Policy is currently experimental and migh
t change or be removed in the future 🎉 Please share your feedback on it in the forum (https://forum.rasa.com) to help u
s make this feature ready for production.
2023-06-28 11:28:25 INFO     root  - Rasa server is up and running.
```

Figura 5.18: Server rasa funzionante

CAPITOLO 6

Manuale Utente

Il presente capitolo è dedicato alla descrizione dettagliata del manuale utente del sistema/chatbot sviluppato. Questo manuale rappresenta una guida essenziale per gli utenti che intendono interagire con il sistema, fornendo istruzioni chiare e dettagliate su come utilizzarlo.

Attraverso questo manuale, gli utenti potranno acquisire familiarità con l'interfaccia del sistema, apprendere le funzionalità disponibili e scoprire i vari comandi e le opzioni a loro disposizione. Saranno fornite indicazioni su come avviare il charbot, su come interagire con esso e come sfruttare al meglio le sue capacità per raggiungere i propri obiettivi. Il manuale utente sarà strutturato in modo intuitivo e suddiviso in sezioni e sottosezioni chiare e ben organizzate, consentendo agli utenti di trovare rapidamente le informazioni di cui hanno bisogno. Verranno inclusi esempi pratici, istruzioni passo-passo e suggerimenti utili per facilitare l'apprendimento e l'utilizzo del sistema.

6.1 Introduzione

Il manuale è stato progettato per fornire tutte le informazioni necessarie per utilizzare efficacemente il chatbot e sfruttarne appieno le funzionalità.

Il chatbot è stato sviluppato con l'obiettivo di offrire un'esperienza di interazione intuitiva e coinvolgente, consentendo agli utenti di ottenere risposte immediate alle loro domande e supporto nelle attività specifiche. Attraverso una combinazione di intelligenza artificiale e linguaggio naturale, il chatbot è in grado di comprendere i messaggi che vengono inviati ad esso e sarà in grado di fornire risposte rilevanti e personalizzate.

Nel corso di questa guida si cercherà di accompagnare l'utente nei vari aspetti dell'utilizzo del chatbot. Si inizierà illustrando le istruzioni di avvio e le modalità di interazione con l'interfaccia utente. Successivamente, verranno mostrate le funzionalità principali del chatbot.

È importante sottolineare che il chatbot è stato progettato per apprendere e migliorare continuamente nel tempo. Ciò significa che sarà in grado di comprendere e rispondere in modo sempre più accurato alle domande fornite man mano che si interagirà con esso.

Questa guida sarà di aiuto nel trarre il massimo dal chatbot sviluppato. Di seguito verrà fornita una spiegazione dettagliata delle varie funzionalità.

6.2 Navigazione dell'Interfaccia

In questa sezione verranno illustrate e spiegate tutte le diverse componenti visive che compongono l'interfaccia del chatbot all'interno di un sito web, fornendo una descrizione dettagliata di ciascuna di esse e del loro significato.

All'apertura del sito internet, l'icona del chatbot sarà visibile in basso a destra della pagina. Nel caso in cui l'icona non appaia, ciò potrebbe indicare un eventuale mancato avvio dei servizi necessari per il funzionamento del chatbot. La Figura 6.1 rappresenta un'immagine illustrativa di come sarà strutturato un sito internet, che includerà il chatbot sviluppato nel presente progetto. In particolare, i punti indicati nella figura corrispondono a:

1. l'indirizzo del sito internet;
2. l'icona del chatbot;
3. il contenuto del sito internet.

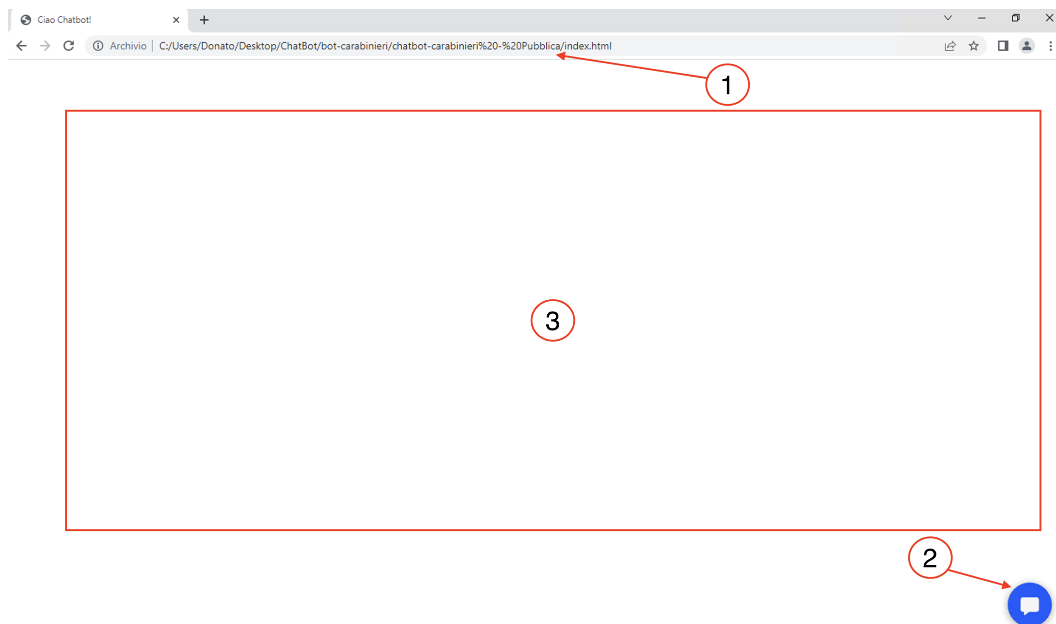

Figura 6.1: Interfaccia del chatbot in un sito internet

Per avviare la conversazione e accedere all'interfaccia del chatbot, è necessario fare click sull'icona corrispondente. Una volta premuta l'icona, si aprirà una piccola finestra con un effetto "saracinesca", offrendo così la possibilità di avviare una conversazione tramite chat. All'interno di questa finestra, saranno visualizzati il nome rappresentativo della chat e una breve descrizione della sua utilità (Punto 1 e 2 della Figura 6.2). In seguito, si noterà, che l'icona del chatbot sarà stata sostituita da una grande "X" (Punto 3 della Figura 6.2). Facendo click su questa icona, la finestra si ridurrà all'icona di partenza, consentendo di mantenere aperta la conversazione, senza doverla chiudere. Infine, dopo pochi secondi dall'apertura della finestra, il chatbot inizierà a mostrare segni di attività, con tre puntini animati, indicando che sta pensando e sta per scrivere (Punto 4 della Figura 6.2). Questi ultimi segnaleranno l'avvio della conversazione e il chatbot inizierà a scrivere un saluto ed una presentazione delle sue funzionalità (Figura 6.3).

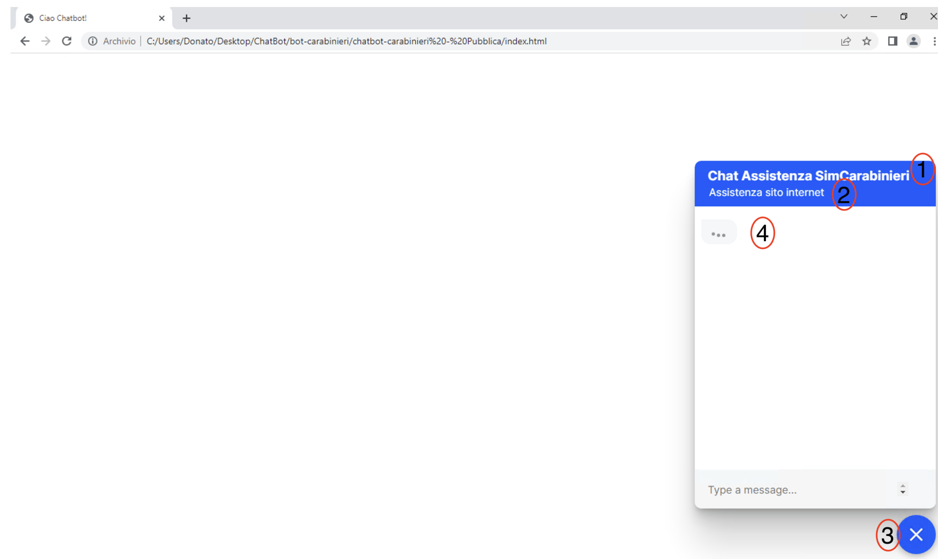

Figura 6.2: Finestra della chat

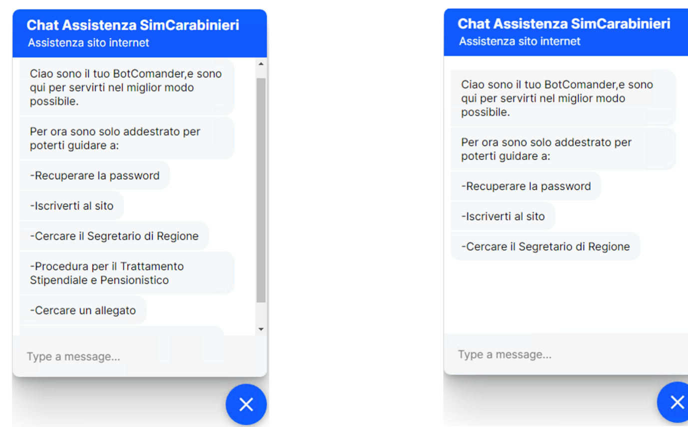

Figura 6.3: Saluto e lista delle funzionalità del chatbot (parte pubblica e parte privata)

Nonostante la conversazione si basi principalmente su messaggi di testo, il chatbot ha la capacità di rispondere anche attraverso pulsanti o immagini (Figure 6.4 e 6.5). Nella Figura 6.4 è possibile osservare che le caselle di testo grigie, posizionate sempre nella parte sinistra della finestra, corrispondono ai messaggi inviati dal chatbot; inceve, le caselle di testo blu sulla parte destra, rappresentano i messaggi inviati dall'utente. Nel caso in cui il chatbot risponda con dei pulsanti, è evidente che nella parte sinistra della finestra verrà visualizzata una lista di azioni in blu, che corrispondono ai pulsanti stessi. Facendo click su uno dei pulsanti, l'azione corrispondente verrà eseguita e il chatbot fornirà una risposta in base all'opzione selezionata (Figura 6.4).

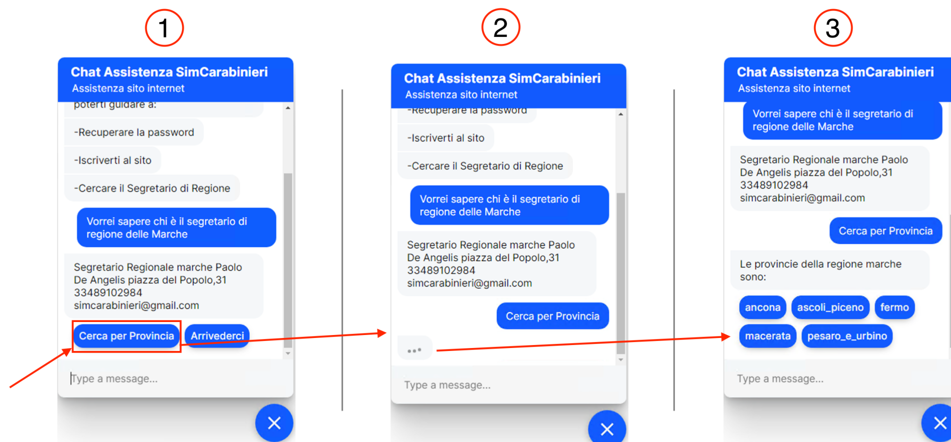

Figura 6.4: Risposta con pulsanti

Figura 6.5: Risposta con immagini

6.3 Utilizzo delle funzionalità

Questa sezione sarà suddivisa in una parte pubblica e una parte privata, facendo riferimento a quanto discusso nel Capitolo 3. Nonostante alcune funzionalità siano presenti sia nella sezione pubblica che in quella privata, si è ritenuto opportuno separare l'illustrazione e la spiegazione delle funzionalità nelle rispettive sezioni, creando due guide distinte.

6.3.1 Funzionalità relative alla Sezione Pubblica

Nella sezione pubblica, sono presenti le seguenti funzionalità:

- l'iscrizione al sito;

- il recupero della Password;
- la ricerca del Segretario di Regione.

Dopo aver richiesto la procedura di iscrizione tramite messaggio, il chatbot fornirà una risposta composta da tre messaggi distinti (Figura 6.6).

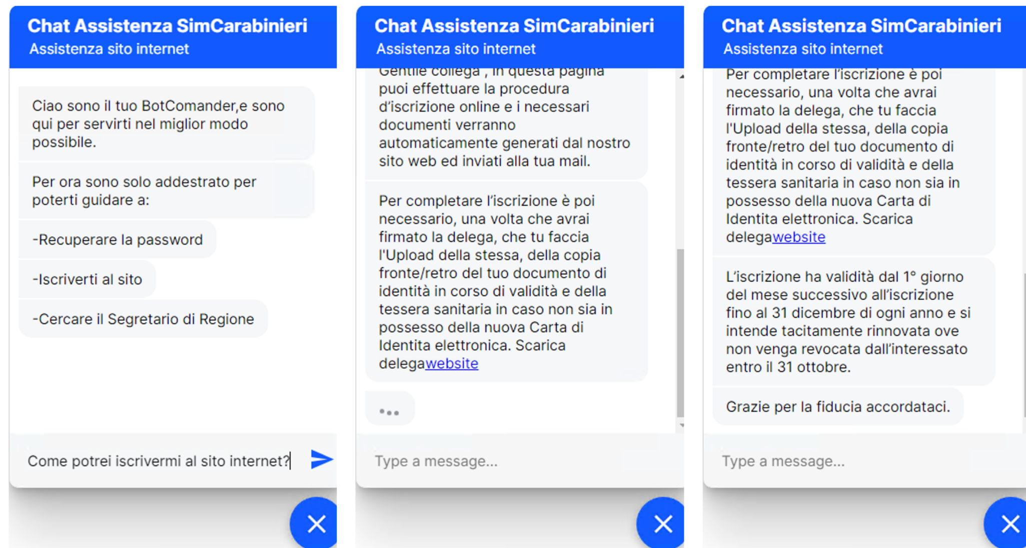

Figura 6.6: Procedura per l'iscrizione al sito

Allo stesso modo, se viene richiesta la procedura per il recupero della password, il chatbot risponderà con sei messaggi distinti (Figura 6.7).

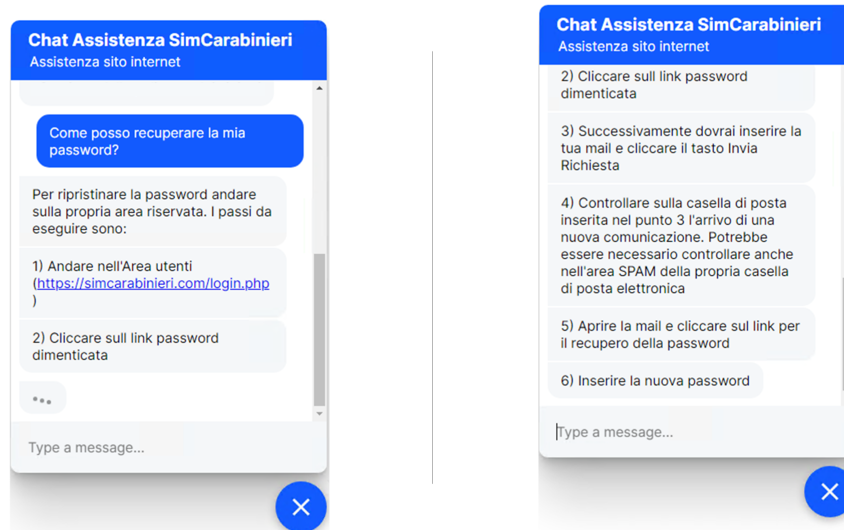

Figura 6.7: Procedura per il recupero della password

Per attivare la funzione di ricerca del Segretario di Regione, esistono diverse modalità, attraverso le quali il chatbot può interpretare il messaggio dell'utente e attivare tale funzione. Le tipologie comprendono:

- scrivere un messaggio generico con all'interno una regione (Punto 1 della Figura 6.8);

- scrivere solo la regione (Punto 2 della Figura 6.8);
- chiedere di ricercare un segretario di regione (Punto 3 della Figura 6.8).

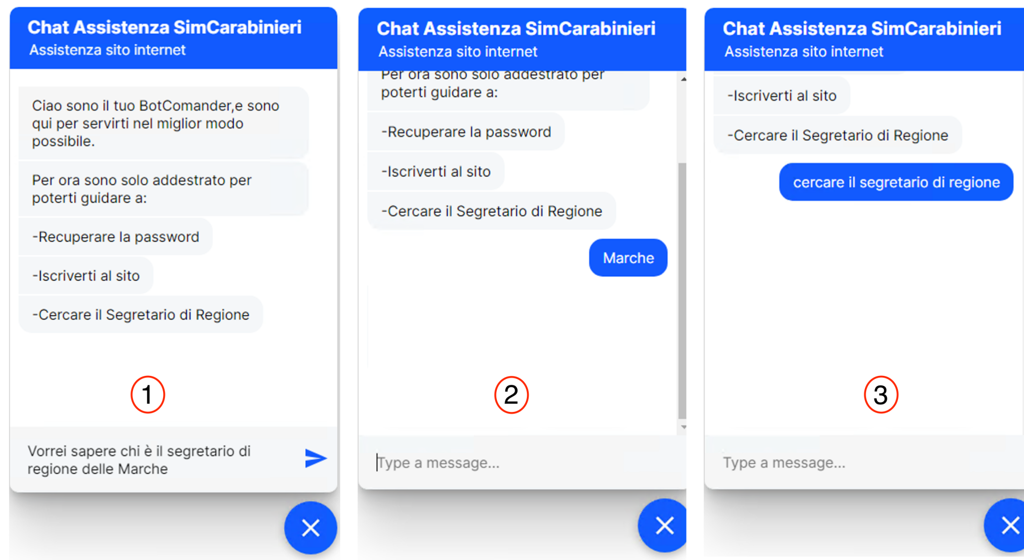

Figura 6.8: Modalità per l'attivazione della procedura della ricerca del Segretario di regione

Per le prime due modalità, il chatbot risponderà fornendo un messaggio, che includerà:

- Nome;
- Cognome;
- Indirizzo;
- Numero di Cellulare;
- E-mail.

Fornite le informazioni, il chatbot invierà un altro messaggio con due pulsanti, chiedendo all'utente se desidera continuare la ricerca per il segretario di Provincia, oppure terminare l'operazione di ricerca, premendo il pulsante con la scritta *Arrivederci* (Punto 1 della Figura 6.9). Se l'utente sceglie di premere il pulsante *Arrivederci*, il chatbot risponderà con un messaggio di saluto (Figura 6.10). Se, invece, l'utente desidera proseguire la ricerca per il segretario di provincia, e preme sul pulsante corrispondente, il chatbot risponderà con un altro messaggio, contenente pulsanti, ciascuno rappresentante una provincia della regione precedentemente cercata (Punto 2 della Figura 6.9). A questo punto, una volta che l'utente seleziona la provincia desiderata, premendo sul relativo pulsante, il chatbot risponderà con messaggi separati, contenenti le informazioni specifiche del segretario di provincia (Punto 3 della Figura 6.9).

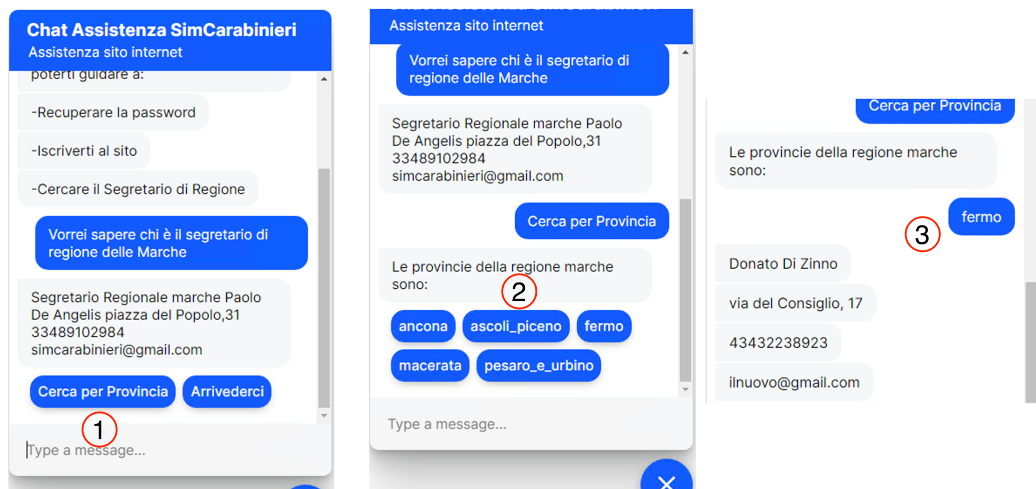

Figura 6.9: Azione per cercare il segretario di provincia

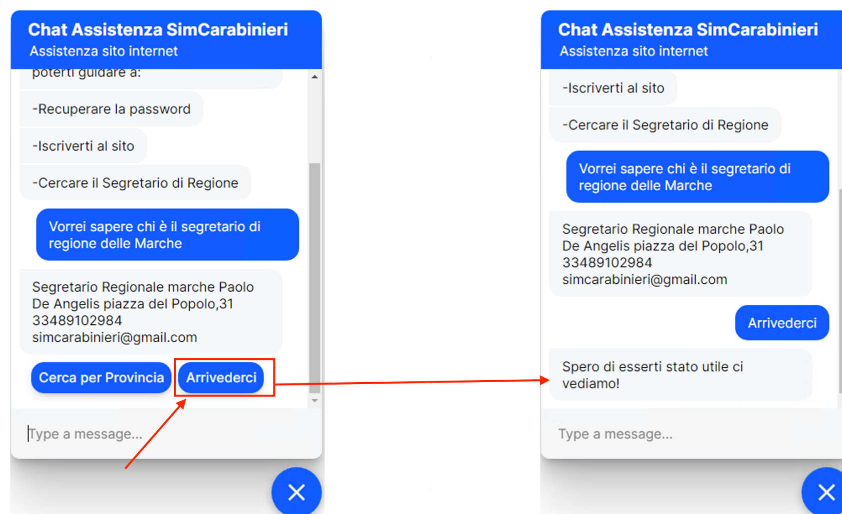

Figura 6.10: Messaggio di *Arrivederci*

Nel caso della terza modalità di attivazione della funzione di ricerca del Segretario di Regione, se il chatbot non rileva una regione specificata nel messaggio di input, ma riconosce l'intenzione dell'utente di cercare il segretario di regione, esso risponderà chiedendo all'utente di inserire la regione del segretario che desidera cercare. Dopo che l'utente ha fornito la regione, il chatbot risponderà con le informazioni pertinenti e, successivamente, chiederà nuovamente all'utente se desidera cercare il segretario di provincia, o terminare la conversazione, premendo il pulsante *Arrivederci* (Fiugra 6.11).

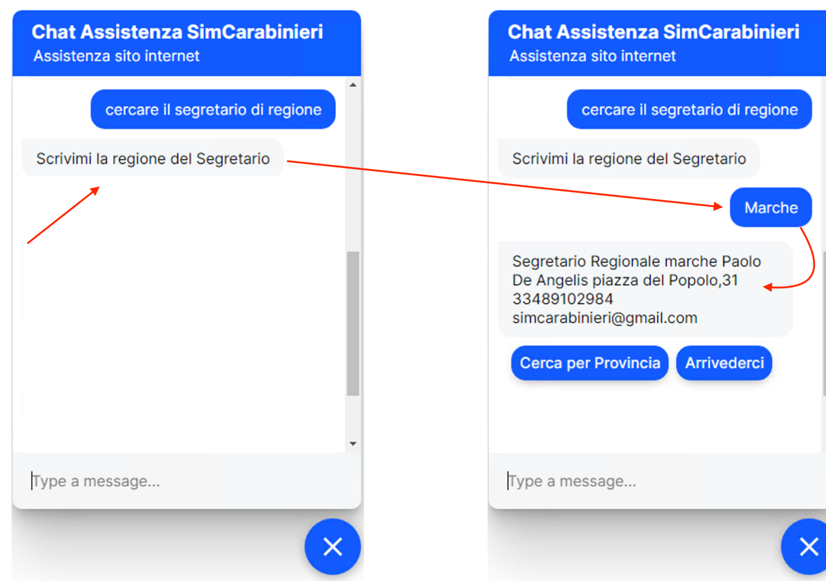

Figura 6.11: Terza modalità per l'attivazione della funzione ricerca segretario di regione

6.3.2 Funzionalità Sezione Privata

Questa sezione si focalizza sull'illustrazione e la spiegazione delle funzioni della sezione privata. Poiché la sezione privata comprende sia le funzioni della sezione pubblica che funzioni specifiche dedicate solo ad essa, le funzioni già trattate nella sezione pubblica non saranno ripetute, ma si invita il lettore a fare riferimento al Paragrafo 6.3.1 per ulteriori dettagli.

Le funzioni specifiche della sezione privata sono:

- procedura per il trattamento Pensionistico e Stipendiale;
- ricerca di un allegato;
- gestione della polizza e dei sinistri.

Per attivare la funzione relativa alla procedura per il trattamento pensionistico o stipendiale, l'utente deve scrivere parole pertinenti a uno dei due trattamenti. Nel caso in cui l'utente desideri avviare la procedura per il trattamento pensionistico, il chatbot fornirà sette messaggi distinti (Figura 6.12). Per quanto riguarda, invece, il trattamento stipendiale, il chatbot risponderà con sei messaggi distinti (Figura 6.13).

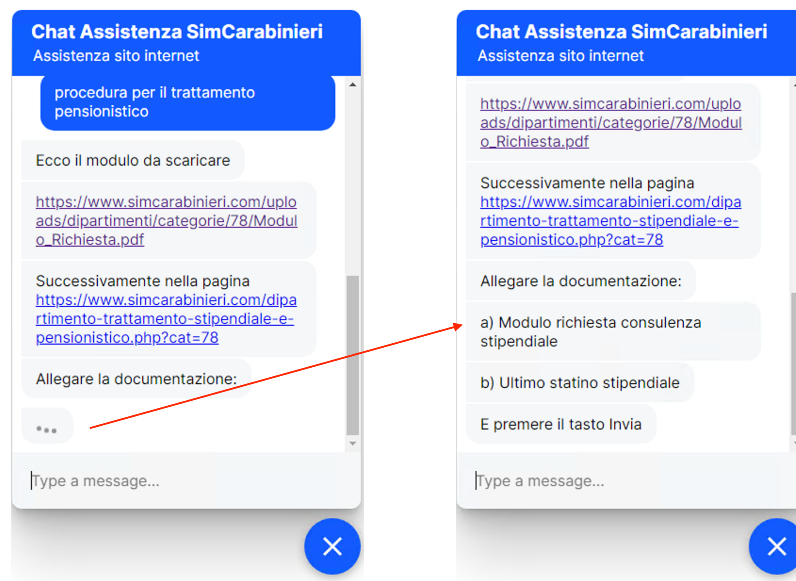

Figura 6.12: Procedura per il trattamento pensionistico

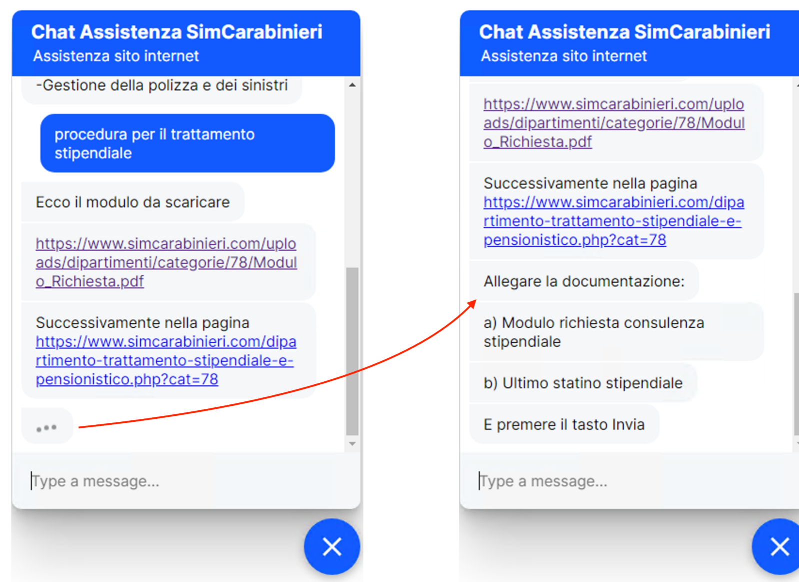

Figura 6.13: Procedura per il trattamento stipendiale

La funzione di ricerca degli allegati può essere richiamata in modo semplice. Per attivarla, basta includere nel messaggio le parole "file", "documento" o "allegato" e il chatbot comprenderà che si desidera effettuare una ricerca all'interno del sito (Punto 3 della Figura 6.14). È stato inoltre previsto che l'utente possa cercare direttamente un allegato utilizzando lo stesso messaggio di attivazione (Punto 1 e 2 della Figura 6.14). Pertanto, nel messaggio, è sufficiente inserire una delle parole chiave menzionate in precedenza, seguita dall'argomento dell'allegato da cercare. Nel caso in cui venga inserita solo la parola chiave, il chatbot attiverà la funzione di ricerca e risponderà con due messaggi distinti. Nel primo messaggio, inviterà l'utente a specificare l'argomento che desidera cercare, mentre nel secondo messaggio lo avviserà di utilizzare la parola *fermati* per interrompere la ricerca (Punto 3 della Figura 6.14). Questa istruzione è fondamentale poiché il chatbot, una volta attivata la funzione di ricerca,

continuerà ad eseguirla in modo indefinito, finché non trova un allegato o l'utente non la interrompa, tramite l'apposita istruzione (Figura 6.15).

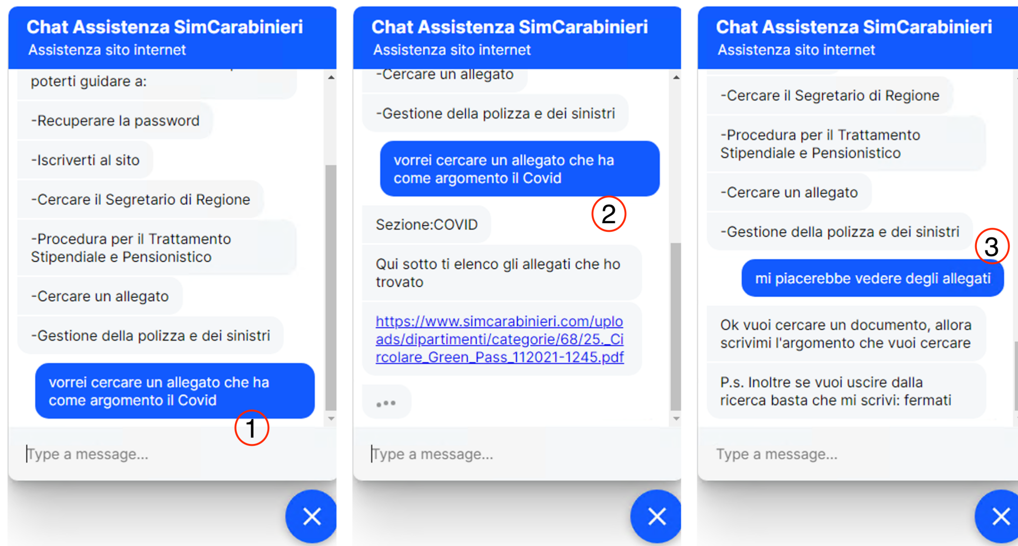

Figura 6.14: Attivazione della funzione di ricerca dell'allegato

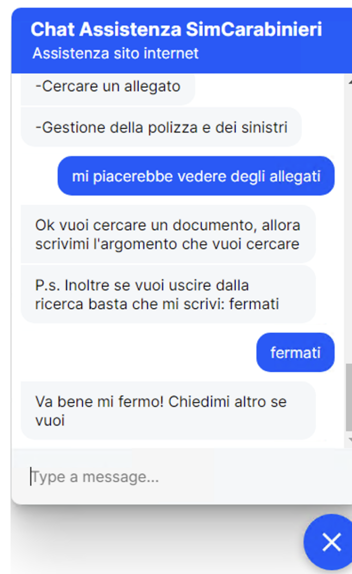

Figura 6.15: Istruzione per fermare la funzione di ricerca dell'allegato

Infine, per attivare la procedura di gestione della polizza e dei sinistri, è sufficiente includere all'interno del messaggio le parole relative alla funzione stessa. Il chatbot risponderà con una serie di messaggi distinti, accompagnati da immagini (Figura 6.16).

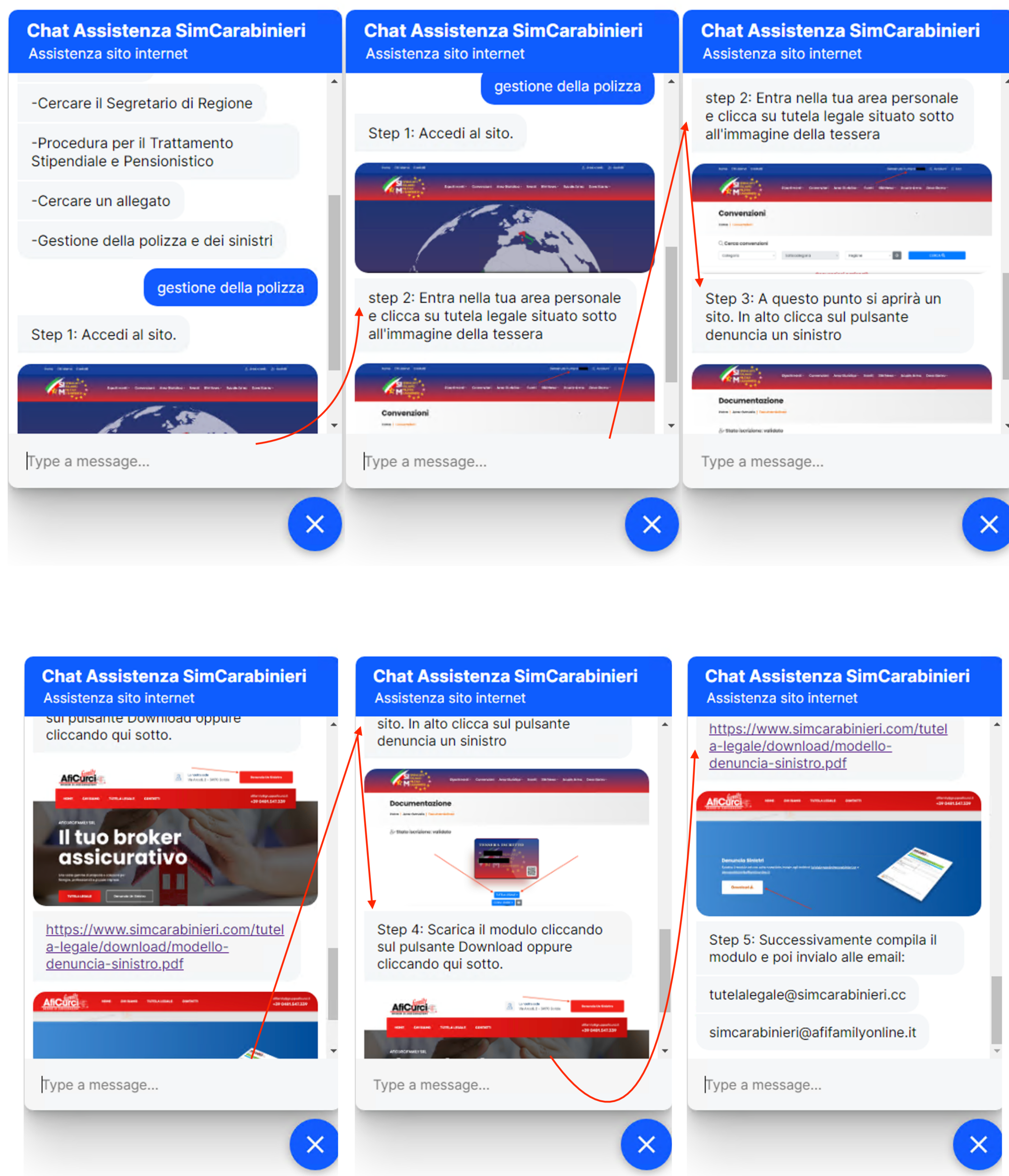

Figura 6.16: Procedura per la gestione della polizza e dei sinistri

6.3.3 Gestione degli errori

Questa sezione è dedicata alla gestione degli errori di interpretazione riguardanti l'intenzione dell'utente da parte del chatbot e alla sua capacità di guidare l'utente verso l'intenzione desiderata. Nel caso in cui il chatbot non riesca a determinare correttamente l'intenzione dell'utente in base al messaggio di input, risponderà con un messaggio specifico: *Attualmente non riesco a gestire questa richiesta. Rincominciamo da capo. Ti do una mano*. Successivamente, saranno presenti una serie di pulsanti che rappresentano le funzioni disponibili. L'utente può selezionare uno dei pulsanti per guidare correttamente il chatbot verso la sua intenzione (Figura 6.17).

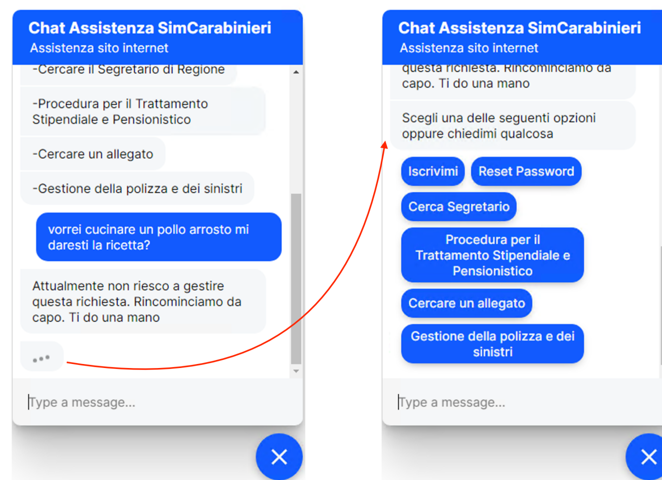

Figura 6.17: Gestione del mancato riconoscimento dell'intenzione

Le modalità con cui il chatbot guida l'utente verso una specifica intenzione saranno esaminate in modo dettagliato, prendendo in considerazione ciascuna di esse singolarmente.

Per quanto riguarda la funzione di ricerca del Segretario di Regione, il chatbot affronterà le seguenti eventuali circostanze:

1. Qualora il segretario della regione ricercata non sia presente, il chatbot risponderà comunicando che il segretario non è ancora stato nominato (Punto 1 della 6.18).

2. Nel caso in cui l'utente cerchi direttamente il Segretario di Provincia, il chatbot risponderà richiedendo all'utente di fornire prima il nome della regione (Punto 2 della 6.18).

3. Nel caso della terza modalità, in cui il chatbot richieda all'utente di inserire la regione del segretario, se l'utente inserisce una qualsiasi altra parola, il chatbot, prima di abbandonare l'intenzione di cercare il Segretario di Regione, chiederà se la parola inviata precedentemente fosse corretta (Punto 3 della 6.18).

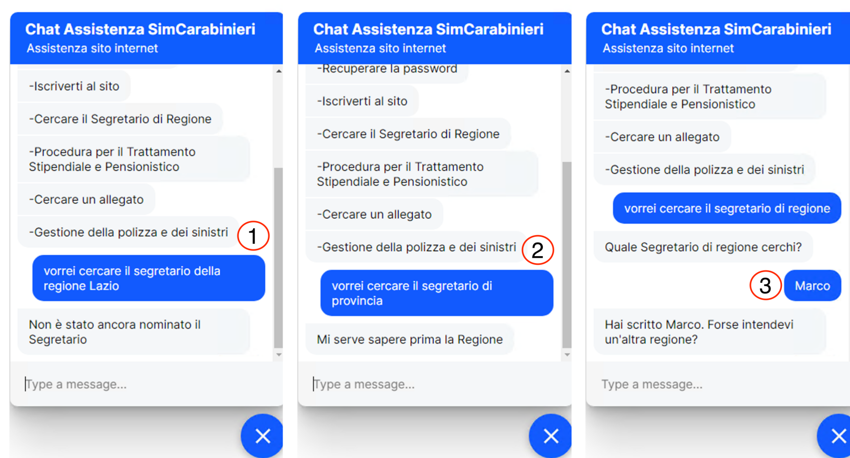

Figura 6.18: Gestione della funzione: ricerca Segretario di Regione

Un'altra funzione che il chatbot gestirà riguarderà la gestione del trattamento Pensionistico e Stipendiale. Nel caso in cui l'utente non specifichi il trattamento nel messaggio e richieda

entrambi, il chatbot risponderà con due pulsanti e chiederà all'utente di selezionare quale dei due trattamenti desidera (Figura 6.19).

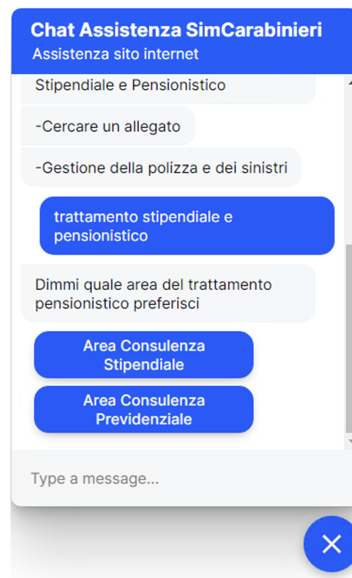

Figura 6.19: Gestione della funzione relativa alla gestione del trattamento Pensionistico e Stipendiale

Infine, l'ultima funzione gestita dal chatbot sarà la ricerca dell'allegato. Nel caso in cui il chatbot individui l'argomento dell'allegato, ma non trovi alcun allegato corrispondente, comunicherà questa informazione all'utente (Figura 6.20).

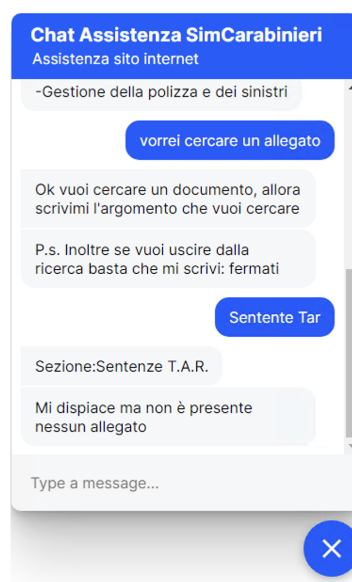

Figura 6.20: Gestione della funzione relativa alla ricerca dell'allegato

CAPITOLO 7

Discussione in merito al lavoro svolto

Nel presente capitolo si fornirà un riepilogo del periodo di tirocinio, analizzando il lavoro svolto.

7.1 Contestualizzazione del lavoro

Nel contesto del lavoro svolto per la creazione di un chatbot dedicato al supporto del Sindacato Italiano dei Carabinieri, è fondamentale fornire una chiara contestualizzazione del progetto. In questo paragrafo si cerca di presentare il contesto in cui il chatbot è stato sviluppato e gli obiettivi specifici della sua applicazione.

Il chatbot è stato creato per fornire supporto ai Carabinieri, al fine di offrire un canale di comunicazione immediato e interattivo, consentendo di accedere a informazioni utili e di poter dare risposte alle domande più frequenti. Grazie a questa soluzione basata sulla tecnologia del linguaggio naturale, i Carabinieri possono porre domande relative a servizi, procedure e altro ancora, ottenendo risposte tempestive ed efficaci.

L'obiettivo principale del chatbot è semplificare e ottimizzare l'interazione tra i Carabinieri e il sito internet, offrendo un supporto immediato e personalizzato 24 ore su 24, 7 giorni su 7. Grazie alla sua capacità di comprendere e rispondere alle domande degli utenti, il chatbot mira a ridurre il carico di lavoro del personale, fornendo nel contempo un servizio efficiente e di qualità.

7.2 Analisi dei risultati

Durante questo progetto, sono state implementate molteplici funzioni per il Sindacato Italiano Militari dei Carabinieri. Dai risultati ottenuti e dalle ottimizzazioni effettuate per garantire l'integrazione corretta del chatbot con il sito web e l'app di messaggistica Telegram, si può affermare che il suo utilizzo può essere considerato soddisfacente e meritevole di attenzione. Questa considerazione deriva dal fatto che, durante la fase di progettazione, si è cercato di rendere il chatbot il più funzionale possibile, in grado di gestire qualsiasi situazione inerente l'ambito. E' stato fatto il possibile per evitare che gli utenti, utilizzando il chatbot, si trovino nell'incapacità di soddisfare le proprie richieste, offrendo loro assistenza in ogni modo possibile per completare la richiesta. Inoltre, per quanto riguarda la funzione di ricerca del segretario, il tempo necessario per ottenere le informazioni è quasi impercettibile per l'utente. Per quanto riguarda la ricerca dell'allegato, invece, si nota una capacità di analisi del

testo dettagliata e precisa, in grado di individuare le parole chiave per la ricerca dell'allegato. Infine, nonostante i risultati promettenti, è possibile affermare con certezza che esiste ancora ampio spazio per miglioramenti futuri, come verrà descritto nel Capitolo 8.

7.3 Valutazione delle prestazioni

In questa sezione si intende fornire una valutazione oggettiva delle prestazioni del chatbot sviluppato. Nonostante le limitate risorse hardware a disposizione per lo sviluppo, si può constatare che le stesse sono state sufficienti per eseguire tutte le funzioni del chatbot, entro tempi accettabili. Infatti, qualsiasi richiesta dell'utente viene soddisfatta in un intervallo di tempo compreso tra uno e tre secondi, al massimo.

L'unica funzione che potrebbe richiedere un tempo superiore a questo range è la ricerca dell'allegato, poiché l'algoritmo sviluppato richiede un elevato numero di permutazioni per trovare l'intersezione dei dati.

Naturalmente, sarebbe possibile ridurre il numero di permutazioni, ma si è scelto di non farlo per preservare l'accuratezza, caratteristica considerata fondamentale. Nonostante tutto, il tempo per ottenere un risultato si aggira generalmente tra i tre e i sette secondi, al massimo.

CAPITOLO 8

Conclusioni e uno sguardo al futuro

Nella presente tesi, sono state illustrate la progettazione e l'implementazione di un chatbot, nonché la sua integrazione su diverse piattaforme. Nella prima parte del lavoro, sono state esaminate le motivazioni che spingono le aziende a includere un chatbot nei loro sistemi. Successivamente, sono stati introdotti gli strumenti software utilizzati e le varie componenti necessarie per la creazione di un assistente virtuale.

Il nucleo centrale della tesi è stato dedicato all'analisi dettagliata dell'argomento principale, ovvero la progettazione del sistema e della logica che costituisce il chatbot. Sono stati considerati gli aspetti e le problematiche da affrontare per garantire il successo nella sua creazione utilizzando algoritmi di elaborazione del linguaggio naturale (NLP) e sviluppando appositi algoritmi per ottimizzarne le funzioni.

Dopo la fase di progettazione, è stata eseguita l'implementazione del chatbot sia su un sistema web che su una delle app di messaggistica più popolari, come Telegram. Per agevolare chi ha commissionato la sua realizzazione, e per consentirne un utilizzo completo, è stato redatto un manuale utente specifico per l'implementazione del chatbot su un sistema web. In conclusione, in questa tesi, sono state affrontate in modo approfondito la progettazione e realizzazione di un chatbot, offrendo un quadro completo delle fasi di sviluppo e mettendo a disposizione le risorse necessarie per l'utilizzo corretto e ottimale dell'assistente virtuale.

Durante l'elaborazione di questa tesi sono stati individuati diversi argomenti meritevoli di approfondimento, che potrebbero arricchire il lavoro svolto finora. Innanzitutto, l'argomento della NLP offre opportunità di ulteriori sviluppi, poiché è in costante evoluzione con aggiornamenti che possono rendere il sistema sviluppato ancora più performante e preciso.

Un altro aspetto degno di nota è il software Rasa, che continua a progredire, introducendo significative novità, tra cui la possibilità di utilizzare il rinomato modello di linguaggio GPT-3.5.

Per quanto riguarda il chatbot già sviluppato, vi sono margini per migliorare ulteriormente le sue funzioni, ad esempio affrontando e gestendo gli errori che non sono stati ancora trattati, e arricchendo ulteriormente i dati di addestramento per potenziare il modello.

Infine, qualora si decidesse di implementare il chatbot nel sito internet del Sindacato Italiano Militari dei Carabinieri, si potrebbe eliminare l'utilizzo dei file come database e, invece, creare una funzione specifica per consentire una connessione diretta al database, migliorando, così, l'efficienza complessiva del sistema.

Bibliografia

AMAZON (2023), *Amazon Lex, IA e Chatbot Conversazionali*. (Cited at page 10)

ANNE KAO, S. P. (2005), «Text mining and natural language processing: introduction for the special issue», .

CRYSTAL JING LUO, D. E. G., VICTOR YIU LUN WONG (2020), «Code Free Chatbot Development: An Easy Way to Jumpstart Your Chatbot!», .

ELLMANN, M. (2018), «Natural language processing (NLP) applied on issue trackers», .

ELLMANN, M. (2019), «Extracting meaning from text and creating a custom language model to optimize NLP results: NLP hands-on workshop series», .

GOOGLE (2023), *Dialogflow*. (Cited at page 10)

HAMISH CUNNINGHAM, Y. W., KEVIN HUMPHREYS (1997), «Software infrastructure for natural language processing», .

HELMA TORKAMAAN, J. Z. (2020), «Exploring chatbot user interfaces for mood measurement: a study of validity and user experience», .

JASPER FEINE, A. M., STEFAN MORANA (2020), «A chatbot response generation system», .

JIEPU JIANG, N. A. (2020), «Response Quality in Human-Chatbot Collaborative Systems», .

MICROSOFT (2023), *Luis, Language Understanding*. (Cited at page 10)

MINGKUN GAO, A. X., XIAOTONG LIU (2020), «Chatbot or Chat-Blocker: Predicting Chatbot Popularity before Deployment», .

MIRELLA LAPATA, F. K. (2005), «Web-based models for natural language processing», .

RASA (2023), *RASA Open source conversational AI*. (Cited at page 10)

SOOMIN KIM, G. G., JOONHWAN LEE (2019), «Comparing Data from Chatbot and Web Surveys: Effects of Platform and Conversational Style on Survey Response Quality», .

WIKIPEDIA (2023), *Apple Silicon*. (Cited at page 22)

Siti web consultati

- Visual Studio Code – `https://code.visualstudio.com/`
- Anaconda – `https://anaconda.org/`
- Rasa – `https://rasa.com/docs/rasa/`
- Google – `https://dialogflow.cloud.google.com/`
- Amazon – `https://aws.amazon.com/it/lex/`
- Microsoft
  – `https://azure.microsoft.com/it-it/services/cognitive-services/conversational-language-understanding/#overview`
- Sindacato Italiano Militari Carabinieri
  – `https://www.simcarabinieri.com/index.php`
- Fermo Tech –`https://fermotech.it/`
- Wikipedia – `https://it.wikipedia.org/`
- Apple Silicon –`https://it.wikipedia.org/wiki/Apple_Silicon`
- UTM – `https://mac.getutm.app/`
- MinHash LSH Ensemble –
  `http://ekzhu.com/datasketch/lshensemble.html`

Ringraziamenti

Il primo ringraziamento è per te, Valentina, amore mio. Hai condiviso gran parte di questa esperienza con me e insieme abbiamo affrontato gioie e dolori. Ti ringrazio perché sei stata la mia fonte di forza e determinazione, hai creduto in me fino alla fine, e senza di te tutto questo probabilmente non sarebbe stato possibile. Ora finalmente possiamo scrivere la parola Fine a questo capitolo della nostra storia. Grazie a te, sono riuscito ad affrontare ogni sfida e ogni ostacolo che ho trovato lungo il mio percorso. Grazie per il tempo che hai investito, non lasciandomi mai solo ed accompagnandomi fino a questo traguardo. Dedico, anzi, regalo questa vittoria a te, perché desidero ripagarti un giorno per tutto quello che hai fatto per me. Vorrei scriverti ancora molte parole, ma per il momento mi fermo qui poichè ciò che desidero dirti appartiene solo a noi e alla nostra storia, non solo a questo capitolo. Grazie di cuore, Ti amo davvero tanto.

Desidero esprimere un ringraziamento speciale ai miei genitori. Grazie a loro e ai loro sacrifici tutto questo è stato reso possibile. Li ringrazio per aver sempre creduto in me, investendo tempo e risorse, e spero di averli resi orgogliosi. Da loro ho appreso le fondamenta della mia vera formazione, che mi ha permesso di diventare la persona che sono oggi e di raggiungere questo obiettivo. So di non essere stato un bambino facile da gestire, e probabilmente ho causato loro preoccupazioni e giorni inquieti. Tuttavia, posso assicurare di aver ascoltato attentamente tutto ciò che mi avete detto. Vi ho osservato con attenzione, cercando di imparare il più possibile. Papà, in particolare, ricordo le tue parole "Usa la testa prima di fare qualcosa!" che mi ripetevi durante il lavoro, e posso dire che probabilmente adesso userò la testa. Vi ringrazio di cuore per tutto e spero che il vostro investimento si riveli un ottimo frutto in futuro.

Un ringraziamento speciale va anche a mia sorella Silvia, che ha sempre creduto in me senza esitazioni, e ha sacrificato parte della sua vita per concedermi il tempo necessario a raggiungere questo obiettivo. Grazie di cuore.

Desidero ringraziare i novelli sposi Nicola e Tatiana. Ma perché non chiamarvi semplicemente amici? Sarebbe riduttivo. Vi ringrazio per essere entrati nella mia vita sin dall'inizio di questo percorso, o meglio, sin da quando ci siamo incontrati in quell'anticamera (gli OFA), prima che le porte di questa avventura si aprissero. Da allora, non ci siamo mai allontanati, tanto che qualche giorno fa sono diventato anche il vostro compare. Voglio ringraziarvi per aver fatto parte di questa preziosa fase della mia vita e per avermi regalato splendidi momenti trascorsi insieme a voi.

Vorrei dedicare un ringraziamento a un amico speciale: Emanuele Longheu. Come affermava un famoso scrittore, "nel bel mezzo del cammin di nostra vita, mi ritrovai in una selva oscura." Proprio in quel momento, in mezzo a quella selva oscura, sommerso da esami, ho avuto il piacere di incontrarti, un vero amico. Ti ringrazio per la tua costante sincerità, disponibilità, lealtà e per quella giusta "durezza" che mi ha aiutato a mantenere la lucidità e a rimanere sulla giusta rotta. Grazie per i preziosi momenti trascorsi insieme all'università, e non solo.

Vorrei ringraziare Giada, per tutti i meravigliosi momenti condivisi lungo questo percorso da ormai dieci anni. Vorrei inoltre ringraziarti per essere stata la persona che mi ha permesso di conoscere la persona più speciale e significativa della mia vita.

Vorrei ringraziare il prof. Ursino, per aver accettato di essere il mio relatore, per essersi speso per me e per la straordinaria disponibilità dimostrata nel corso della stesura di questa tesi. In ultimo, desidero sinceramente ringraziarla per il tempo che mi ha dedicato nel fornirmi consigli preziosi e nel guidarmi nelle decisioni che ho dovuto prendere.

Ringrazio il mio correlatore, per la sua continua disponibilità e il suo prezioso supporto durante il mio tirocinio.

Vorrei ringraziare tutti i miei amici, per essere stati al mio fianco e per avermi permesso di trascorrere parte della mia vita nel miglior modo possibile.

Vorrei dedicare un piccolo ringraziamento anche a Odoardo Recchi, che purtroppo non è più con noi in questo momento. Vorrei ricordarlo per gli insegnamenti preziosi che mi ha donato e per aver contribuito a colmare alcune delle mie lacune.

Si è concluso un capitolo importantissimo, e ne sta per cominciare uno nuovo. Un grande grazie alla mia Università, comprendendo tutto ciò che ha incluso, per avermi dato l'opportunità di essere qui, ora.